BERGISCHE
UNIVERSITÄT
WUPPERTAL

IeeeCC754++
An Advanced Set of Tools to Check IEEE 754-2008 Conformity

Matthias Hüsken

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

Vorgelegt und genehmigt an der
Fakultät für Mathematik und Naturwissenschaften
der Bergischen Universität Wuppertal

Gutachter:
Prof. Dr. Andreas Frommer
Prof. Dr. Dirk Pleiter

Prüfungskommission:
Prof. Dr. Andreas Frommer
Prof. Dr. Dirk Pleiter
Prof. Dr. Bruno Lang
Prof. Dr. Michael Günther

Wuppertal, 06.12.2017

Die Dissertation kann wie folgt zitiert werden:

urn:nbn:de:hbz:468-20180213-104430-5
[http://nbn-resolving.de/urn/resolver.pl?urn=urn%3Anbn%3Ade%3Ahbz%3A468-20180213-104430-5]

Danksagungen

Am Anfang dieser Arbeit möchte ich die Gelegenheit nutzen, mich bei denen zu bedanken, die mich über die Jahre begleitet haben und ohne die es nicht zur Entstehung dieses Werkes gekommen wäre. Mein Dank gilt zuerst Prof. Dr. Andreas Frommer für die Möglichkeit, diese Arbeit zu verfassen, für seine Betreuung und die wertvollen Hinweise sowie die angenehme Atmosphäre in seiner Arbeitsgruppe. Darüber hinaus ermöglichte er mir die Mitarbeit an verschiedensten interessanten Projekten, allen voran QPACE und QPACE3. Ebenso danke ich Prof. Dr. Dirk Pleiter für die wertvollen Anregungen zu dieser Arbeit, die langjährige gute Zusammenarbeit im QPACE-Projekt und das unkomplizierte Zurverfügungstellen einer Reihe von Testplattformen. Prof. Dr. Bruno Lang danke ich vor allem für die zahlreichen inspirierende Unterhaltungen jedweder Art.

Dank netter (ehemaliger und aktueller) Kollegen durfte ich in der Arbeitsgruppe Angewandte Informatik in angenehmer Umgebung arbeiten. Namentlich nennen möchte ich hier Brigitte Schultz, ohne die diese Arbeitsgruppe schon längst zusammengebrochen wäre, Katrin, Holger, Peter, Sonja, Marcel, Jan, Claudia, Artur und Martin. Vielen Dank auch an Willi für die Bereitstellung weiterer Testplattformen sowie an Sonja, Martin, Sarah, Jared und meinen Vater für das Korrekturlesen und die Hilfe bei diversen LaTeX- und Englisch-Fragen.

Weiterer Dank gilt meinen Freunden, mit denen ich die Abenteuer Musik (new challenge) und Kirche (NORDSTERN.kirche) erleben durfte und darf: Alex und Heike, Andre und Anne, Simon und Bine, Gary und Nic, Gerrith und Miri (und meinen wunderbaren Patenkindern Noah, Leah und Ewah), Sontka, Stephan und Verena, Kris und vielen weiteren.

Vielen Dank meiner Familie: meinen Geschwistern und deren Ehepartnern und Kindern, und vor allem meinen Eltern, ohne deren vielfältige Unterstützung es diese Arbeit nicht gegeben hätte.

Jen – du bist das Beste, was mir passieren konnte. Danke für alles!

Und das Wichtigste zum Schluss – meinem Gott: „Denn von ihm und durch ihn und für ihn sind alle Dinge.“ (Die Bibel, Römerbrief, Kapitel 11, Vers 36)
Soli deo gloria.

Contents

Introduction

The "IEEE Standard for Binary Floating-Point Arithmetic" IEEE 754 has arguably been one of the most influential standards for the broad area of scientific computing. Since its publication in 1985, it is the benchmark for the quality of floating-point implementations of processors and floating-point units. Floating point arithmetic conforming to the standard implies that numerical algorithms behave identically across the various computing platforms (or at least in a predictable manner) and that developers of numerical algorithms can rely on the standard when addressing stability issues that might otherwise depend on the floating-point environment. The standard thus facilitates the portability of numerical algorithms across platforms.

With this thesis, we provide a contribution to evaluating the conformity of a given floating-point environment to the latest valid version of the standard. In particular, we extended the well known testing tool `IeeeCC754` with a large number of features, such as support for the current incarnation of the standard called IEEE 754-2008, new analysis facilities, additional floating-point operators, and new test vectors to verify that these operators are implemented in a conforming manner. Furthermore, we heavily expanded the selection of supported floating-point environments and provide facilities to easily extend our new tool `IeeeCC754++` with new ports targeted at future floating-point platforms. We meticulously documented the process of adding a new architecture port and demonstrate its applicability with a port for the ARM SVE floating-point unit (for which hardware is not yet available). Additionally, we developed a testing framework that enables large-scale evaluation of floating-point environments, the graphical application `IeeeCC754++LogViewer` that provides convenient access to the generated logfiles, and a variation of the testing framework tailored to studying the influence of compiler options on the behaviour of numerical applications regarding floating-point accuracy as well as application performance.

Former testing tools including `IeeeCC754` serve a rather technical purpose, i. e. they enable developers and implementers of floating-point units (be it hardware units or software libraries) to verify proper functionality of their implementation. This use case is also supported by `IeeeCC754++`; however, our tool takes a more user centric point of view by enabling the average researcher with the possibility to check her default floating-point environment (which is called default user environment in this thesis) for IEEE 754-2008 conformity.

The thesis is organised as follows: In Chapter 1, we give a short introduction of floating-point numbers and their properties and take a look at the different incarnations of IEEE 754-2008. Afterwards, we define the term user environment and discuss the components that constitute such an environment. Chapter 2 presents former work in providing tools for testing the IEEE-conformity of a floating-point implementation. In particular, we describe `IeeeCC754` which represents the tool that `IeeeCC754++` is based upon, discuss its testing model, and briefly summarise results from former conformity testing.

Chapters 3 to 5 constitute the main part of this thesis in which we describe our contributions to IEEE-conformity testing. Chapter 3 starts with a more detailed discussion of testing strategies and our definition of a user environment, followed by an in-depth look at the new features which we implemented in `IeeeCC754++`. Additionally, we discuss different testing modes targeted at a range of applications such as default user environment testing or IEEE-conformity evaluation of specific floating-point units. We introduce the evaluation and optimisation frameworks which support large-scale testing and studying the influence of compiler options, and present a few additional tools that ease the analysis of the resulting logfiles such as the graphical application `IeeeCC754++LogViewer`. Chapter 4 starts with a description of the floating-point operators which were not part of the original IEEE 754 standard, but which were introduced with IEEE 754-2008, e. g. a fused multiply-add operator and a range of elementary operators such as trigonometric functions or exponentials. We then describe the test vectors which we added to the original `IeeeCC754` testsets in order to check the new floating-point operators. The discussion of the numerous architecture ports which we implemented in `IeeeCC754++` is the subject of Chapter 5. We describe the various underlying classes of platforms such as hardware floating-point units of different processors, accelerators, or software libraries, give a short overview of the history and the corresponding instruction sets or implementations of the respective platforms, and finally discuss their integration into our testing tool `IeeeCC754++`.

In order to demonstrate the use of our contributions regarding IEEE-conformity testing, Chapter 6 presents results for a selection of the architecture ports described in Chapter 5. We also show how to apply the extended tools such as the evaluation and optimisation frameworks or `IeeeCC754++LogViewer` during the testing process. We conclude this thesis by summarising the contributions presented in this thesis and discussing further possible research areas.

Since the extension with further architecture ports is an integral part of our contributions to IEEE-conformity testing, we add three corresponding appendices to this thesis. Appendix A discusses the structure of the `IeeeCC754++` source code together with the newly written build system and describes the process of building and running `IeeeCC754++` executables. In Appendix B, we give detailed instructions on how to add a new architecture port to `IeeeCC754++`. Finally, Appendix C lists some reference material such as the command line options of `IeeeCC754++`, the error codes used within `IeeeCC754++`, and reference files for the evaluation and optimisation frameworks.

Chapter 1

Floating-point numbers, standards, and the user environment

The basics of floating-point arithmetic as an approximation for real-valued arithmetic can be introduced with a simple description in just a few words: Given some base β, a number x is represented with a sign S_x, a significand s_x, and an exponent e such that

$$x = S_x \times s_x \times \beta^e. \tag{1.1}$$

However, making such an arithmetic reliable, fast, and portable is far from simple. Between the 1960s and the 1980s, vast numbers of different floating-point arithmetics were designed and implemented on a wide range of computing platforms, all of them handling issues like binary formats, rounding, overflow, and underflow, in individual and often incompatible manners. Porting numerical code to another platform and verifying the correctness of the resulting program was a highly complex and time-consuming process. This situation led W. Kahan to write his article "Why do we need a floating-point standard?" [Kah81] in which he underlines the need for a common standard that could be both useful for programmers and practical for implementers. Furthermore, he explains the rationale behind the standard draft which was proposed in 1981 after three years of work in the corresponding IEEE subcommittee. It still took another four years of work and discussions to finally release the standard as IEEE 754 in 1985.

In this chapter, we introduce the basic ingredients for the following chapters: floating-point numbers and some of their basic properties, the main IEEE standard concerning floating-point numbers (IEEE 754-2008 and its former incarnations), and what we call the "user's environment": the programming and execution environment a programmer experiences when writing numerical code using floating-point numbers on a given platform. Furthermore, we take a look at what is actually available to the programmer in a given user environment (on some hardware platform).

1.1 The foundation: Floating-point numbers

As [Mul+10, p. xv] notes, "floating-point arithmetic is by far the most widely used way of approximating real-number arithmetic for performing numerical calculations on modern computers." Consequently, a vast selection of material on floating-point arithmetic has been written, including excellent introductions and comprehensive treatises. To avoid giving yet another introduction to floating-point arithmetic, we only introduce floating-point numbers and their features as far as they are needed in this thesis, and refer to [Gol91] and e. g. [Mul+10] for more detailed treatise. As `IeeeCC754++` is based on `IeeeCC754`, most of the notation used in this thesis is borrowed from [VCV01a] and [VCV01b].

Let $\mathbb{F}(\beta, t, L, U)$ be the set of floating-point numbers in base $\beta \geqslant 2$, precision $t > 0$, and exponent range $[L, U]$ with $L < U$. Every floating-point number $x \in \mathbb{F}(\beta, t, L, U)$ consists of a sign $S_x \in \{-1, 1\}$, a significand[1] $s_x = x_0.x_1x_2 \ldots x_{t-1}$ (which has t digits), and an exponent $e_x \in [L, U]$. In this thesis, we will only consider the case $\beta = 2$.

Floating-point representations of a number $x \in \mathbb{F}(\beta, t, L, U)$ are not necessarily unique: For example, both 0.01×10^1 and 1.00×10^{-1} represent the number 0.1 in $\mathbb{F}(10, 3, -1, 1)$. Setting $x_0 \neq 0$ makes the representation of x unique and is called normalisation of x. For $\beta = 2$, this is equivalent to setting $x_0 = 1$. This means that all floating-point numbers in $\mathbb{F}(2, t, L, U)$ with exponent $e_x \in [L, U]$ are normalised, i. e. $x_0 = 1 \Rightarrow s_x = 1.x_1x_2 \ldots x_{t-1}$.

The value of a normalised floating-point number $x \in \mathbb{F}(2, t, L, U)$ is then given by

$$\begin{aligned} x &= S_x \times s_x \times \beta^{e_x} = S_x \times \sum_{i=0}^{t-1} x_i 2^{e_x - i} \\ &= S_x \times 1.x_1x_2 \ldots x_{t-1} \times 2^{e_x}. \end{aligned} \tag{1.2}$$

One must be aware of one drawback of using normalisation: 0 cannot be expressed as a normalised number. This can be solved by reserving a special exponent to represent 0. The most common choice (and that of IEEE 754) is $e_x = L - 1$; see page 9.[2]

Given fixed precision and exponent range, $\mathbb{F}(\beta, t, L, U)$ obviously contains only a finite number of floating point numbers. $\mathbb{F}(\beta, t, L, U)$ is not closed, i. e. if

[1] The significand has also been called mantissa.

[2] Usually, $e_x = 0$ is chosen, resulting in $0 < L < U$. This is mainly due to technical reasons: In most cases, the exponent is encoded as an integer bitstring. To allow for negative exponents, common approaches like two's complement representation are avoided for performance reasons. Instead, an appropriate *bias* is chosen that divides the bitstring into positive and negative numbers and that can be decoded by using simple and therefore fast bit shifts. For details and further reasons for choosing $L - 1 = 0$, see page 9.

$x, y \in \mathbb{F}(\beta, t, L, U)$, the result $z = x \mathbin{\triangle} y$ of an operation $\triangle$ does not necessarily lie in $\mathbb{F}(\beta, t, L, U)$ for one of the following reasons:

- z is too large (in magnitude) to be represented, i. e. $z > \sum_{i=0}^{t-1}(\beta - 1)\beta^{U-i}$. This is called *overflow*.
- z is smaller (in magnitude) than the smallest representable number in $\mathbb{F}(\beta, t, L, U)$, i. e. $0 < |z| < (\beta - 1) \times \beta^L$. This is called *underflow*; z is then called *subnormal* or *tiny*.[3]
- z needs more than t digits to be represented. This might be the case because z needs an infinite number of digits (consider e. g. transcendental numbers like π or periodic numbers like $\frac{1}{3}$ for $\beta = 10$ or 0.1_{10} for $\beta = 2$) or because the representation of z in base β is finite, but z lies between two numbers representable in $\mathbb{F}(\beta, t, L, U)$. In the latter case, z effectively belongs to $\mathbb{F}(\beta, r, L, U)$ for some $r > t$.

To implement a closed version $\big(\mathbb{F}^*(\beta, t, L, U), \circledcirc\big)$ of $\big(\mathbb{F}(\beta, t, L, U), \triangle\big)$, all three of the mentioned cases need to be addressed:

- Overflow might naively be handled by setting z to the largest representable normalised number. However, this is dangerous as it might introduce an error of arbitrary magnitude. More commonly, some kind of exception or a special value to represent ∞ (or both) are utilised to notify the user of potential problems and provide some meaningful insight. A common choice for a representation of a special value is using an exponent of $U + 1$.
- In case of underflow, the easiest approach is to simply set the significand of the resulting value to zero (i. e. $z = 0$, called *flush to zero*). This introduces an error of magnitude $\mathcal{O}(\beta^L)$.

  Another approach is to use the special exponent $L - 1$ (also commonly used for representing 0, see above) to represent z without normalising the significand, thus making use of all potential values of the significand. This fills the gap between 0 and the smallest representable (normalised) floating-point number with evenly spaced numbers, effectively decreasing the error to a magnitude of $\mathcal{O}(\beta^{L-t+1})$.

  In both approaches, an additional exception might be raised to notify the user.
- In the case of z lying between two (adjacent) floating-point numbers $z_1, z_2 \in \mathbb{F}(\beta, t, L, U)$, z must be *rounded* to either x or y. For a discussion of rounding strategies, see page 10. The error introduced by rounding is of magnitude $\mathcal{O}(\beta^{1-t})$. More precisely, the following holds for the relative error $\epsilon(x)$:

[3] Historically, z has also been called *denormalised*.

$\epsilon(x) \leqslant \frac{1}{2}\beta^{1-t}$ in round to nearest mode, and $\epsilon(x) \leqslant \beta^{1-t}$ for the other rounding modes, cf. Section 1.3 and [Mul+10, pp. 23 sq.].

For actual implementations, the issue of illegal operations has to be handled. Given $x \in \mathbb{F}(\beta, t, L, U)$, consider e. g. $x/0$ or $\sqrt{x}$ for $x < 0$ whose result is not defined. Common approaches are to raise an exception and/or to return a special value (usually called *NaN* = Not a Number) to notify the user that no proper floating-point number can be returned. Furthermore, the propagation of values like NaN or ∞ (if used) has to be considered. In the next section, we discuss the way IEEE 754-2008 handles these issues.

In the following, we will denote the extended set of floating-point numbers with base $\beta \geqslant 2$, precision $t > 0$, and exponent range $[L-1, U+1]$, $0 < L < U$, by $\mathbb{F}^*(\beta, t, L, U)$, which includes the special exponents $L-1$ and $U+1$ to represent values of 0, ∞, and NaNs.

1.2 The standards: IEEE 754 and 854

In this section, we present the requirements that the "IEEE Standard for Floating-Point Arithmetic" IEEE 754-2008 demands from conforming floating-point implementations. Starting with the first revision of this standard, IEEE 754 [IEEE85], we present the main principles and concepts of this standard which introduced strict requirements for conforming floating-point implementations for the first time. In order to understand the state of some current implementations and their conformity to the (current) standard, we then discuss the small additions done in the second revision, IEEE 854 [IEEE87], and in more detail the current incarnation of the standard, IEEE 754-2008 [IEEE08]. Finally, we briefly comment on the revision that is currently ongoing to produce IEEE 754-2018.

1.2.1 IEEE 754

When IEEE 754 was released as "IEEE Standard for Binary Floating-Point Arithmetic" in 1985 after years of work, it stated its rationale in the foreword: "This standard defines a family of commercially feasible ways for new systems to perform binary floating-point arithmetic. The issues of retrofitting were not considered." [IEEE85] It aims at providing a sane and understandable way of performing floating-point operations in a well-defined floating-point arithmetic, while at the same time enabling this arithmetic to be implemented in a cost-efficient manner. And, as [Mul+10] notes, it indeed "was a key factor in improving the quality of the computational environment available to programmers."

Nowadays, virtually all processor designs, compilers, and mathematical libraries implement IEEE 754 (at least to some extent) and provide for a predictable computing environment for programmers designing numerical software.

In the following, we explain how IEEE 754 addresses the issues raised in the previous section.

Representations

IEEE 754 starts by specifying two binary number formats for floating-point calculations, the so-called "basic formats" single precision and double precision. Conforming implementations must implement at least the single format.

Furthermore, it defines two extended formats which can be used for intermediary computations to achieve better accuracy compared to when using target precision in intermediary computations. The standard recommends the use of an extended format for the widest implemented precision only. In practice, this means that most platforms implement single, double, and double-extended precision.

The basic formats make use of the base being $\beta = 2$: The first digit of every normalised number's significand is always equal to 1, whereas the first digit of a subnormal number is 0. Given a way to distinguish between normalised and subnormal numbers without looking at the significand (which is possible in IEEE formats, see below), this first digit needs not be explicitly stored but is implied. Due to this clever implementation, the length of the significand is increased by one bit. The implicitly stored bit is called *hidden bit*.

Given a floating-point number $x = S_x \times s_x \times e_x \in \mathbb{F}^*(2, t, L, U)$, IEEE 754 reserves the extremal values $L - 1$ and $U + 1$ of the exponent e_x for special values:

- If $e_x = L - 1$ and the significand $s_x = 0$, then x is a *signed zero* (depending on the value of the sign S_x).
- If $e_x = L - 1$ and $s_x \neq 0$, then x is a *subnormal number*.
- If $e_x = U + 1$ and $s_x = 0$, then x is *infinite*, i. e. $x = S_x \cdot \infty$.
- If $e_x = U + 1$ and $s_x \neq 0$, then x is not a number, i. e. it is an *NaN*.

Additionally, the exponent is stored as a *biased exponent* with bias b, meaning that, when interpreting the stored exponent $E_x \in [0, 2^q - 1]$ of q bits length as an unsigned integer, the value e_x of the exponent is interpreted as follows:

- If $E_x = 0$ $(= L - 1)$, then x is a signed zero or a subnormal number.
- If $E_x = 2^q - 1$ $(= U + 1)$, then x is an infinity or an NaN.
- If $0 < E_x < q - 1$, then x is a normalised number with exponent $e_x = E_x - b$.

The bias is then given by $b = 2^{q-1} - 1$.

With this floating-point encoding, 0 is represented by a bitstring only containing zeroes (or by a bitstring with a leading one and then followed by all zeroes in case of a negative 0), and it is easy to distinguish between normalised and subnormal

numbers, zeroes, infinities, and NaNs. Furthermore, using a biased encoding for the exponent makes it possible to represent positive and negative exponents without using two's complement or sign-magnitude representations, which would have made comparing floating-point numbers harder. Another advantage is that positive floating-point numbers including $+0$ and $+\infty$ are ordered like their binary representation (interpreted as integers). This makes it possible to obtain the next floating-point number (i. e. the floating-point successor) by interpreting the binary representation of a floating-point as an integer and adding one to that integer.

Given these representations, IEEE 754 defines the following formats:

- *Single precision*: 32 bits wide, with 1 sign bit, 8 bits for the exponent, and 23 bits for the significand. This means the following values are used: $t = 24$ with hidden bit, $L = -126$, $U = 127$, $b = 127$, i. e. the standard implements $\mathbb{F}^*(2, 24, -126, 127)$.

- *Double precision*: 64 bits wide, with 1 sign bit, 11 bits for the exponent, and 52 bits for the significand. This means the following values are used: $t = 53$ with hidden bit, $L = -1022$, $U = 1023$, $b = 1023$ i. e. the standard implements $\mathbb{F}^*(2, 53, -1022, 1023)$.

- *Single-extended*: $p \geqslant 32$, hidden bit optional, $L \leqslant -1022$, $U \geqslant 1023$.

- *Double-extended*: $p \geqslant 64$, hidden bit optional, $L \leqslant -16382$, $U \geqslant 16383$.

As a side note, we give the definition of the most widely used double-extended format which is implemented in Intel's x87 floating-point unit (cf. Section 5.2.1):

- *Intel x87 double-extended*: 80 bits wide, with 1 sign bit, 15 bits for the exponent, and 64 bits for the significand. This means the following values are used: $t = 64$ without hidden bit, $L = -16382$, $U = 16383$, $b = 16383$ or $\mathbb{F}^*(2, 64, -16382, 16383)$.

Rounding modes

As discussed in Section 1.1, the result of an operation involving floating-point numbers does not necessarily lie in the set of representable floating-point numbers in the target format. If the precise result is a valid real number, it can be *rounded* to a permissible floating-point number: "Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format" [IEEE85]. IEEE 754 defines four rounding modes: the default rounding mode *round to nearest* and three directed rounding modes *round toward* $+\infty$, *round toward* $-\infty$, and *round toward 0*:

- In round to nearest mode, "the representable value nearest to the infinitely precise result shall be delivered". If the magnitude of the precise result is at

least $2^U(2-2^{-t})$, an infinity with no change in sign is returned. If the precise result lies exactly between two representable floating-point numbers, the number with the least significant bit zero is returned (tie to even). Therefore, this rounding mode is also called *round to nearest even*.

- In the three other modes, the number closest to and no less (round toward $+\infty$), no greater (round toward $-\infty$), or no greater in magnitude (round toward zero) than the infinitely precise result is returned.

Operations

IEEE 754 defines a set of operations on floating-point numbers and mandates that conforming implementations return correctly rounded values for these operations. Correctly rounded means that each operation must be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then this intermediate result is rounded to the target format according to the currently selected rounding mode. Care has to be taken to cover ± 0, infinities, and NaNs; furthermore, IEEE 754 defines "exceptions caused by exceptional operands and exceptional results", see below.

The operations can be categorised into the following groups:

- Arithmetic operations: These include addition, subtraction, multiplication, division, remainder, and square root. In the following, they are also referred to as *basic operations*.

- Conversion operations. The following conversions must be supported:

  - Conversions between all supported floating-point formats. Conversion to a wider format is always exact; conversions to a narrower format must be rounded.

  - Conversion operators between all supported floating-point and integer formats must be provided.

  - Rounding a floating-point number to integral value: A given floating-point number is rounded so that it is exactly convertible to an integer. It is stored in the same floating-point format as the original number.

  - Conversion between binary numbers and decimal representations (also known as binary to decimal conversion and vice versa): To interpret numerical values given as binary floating-point numbers or to input numerical values into a floating-point program, a conforming implementation must provide conversion routines between the supported floating-point formats and a decimal representation. At the time of releasing the standard, efficient algorithms to perform these conversions were not yet known (see [Mul$^+$10, pp. 43 sqq.]). Therefore, IEEE 754

mandates that these conversions are to be correctly rounded only for certain ranges of numbers. For details, see [IEEE85, pp. 7 sq.].

- Comparisons: It must be possible to compare floating-point numbers in all supported formats. Comparisons are always exact. To enable handling of NaNs when comparing floating-point numbers, IEEE 754 introduces the relation *unordered* to describe comparisons between NaNs and any other floating-point number. The usual relations like equal, greater than, equal greater than etc. are available. The sign of a zero is ignored when comparing.

When one of the operands is an infinite value or an NaN, special rules apply:

- Infinities are interpreted as limiting cases for real arithmetic, i. e. $-\infty < x < \infty$ with $x \in \mathbb{F}(\beta, t, L, U)$. Arithmetic on infinite operands is performed in this sense; it is always exact.

- IEEE 754 distinguishes between two types of NaNs: signalling and quiet NaNs. If an operand is a quiet NaN, it is propagated by all operations. If an operation encounters a signalling NaN, the invalid exception (see below) is signalled and a quiet NaN is returned.

It should be noted that a few additional "helper" functions are recommended, such as `copy`, `negate`, `abs`, and `class`. As these only serve more technical purposes and are not influenced by rounding or exceptions, they are of no further relevance in this thesis.

Exceptions and traps

When executing operations on floating-point numbers, it is not always possible that the result of the given operation yields a floating-point number representable in the target format or that it is a number at all. Consider e. g. division by 0 or computing a square root of a negative number. In order to enable the user to appropriately handle exceptional circumstances arising during a floating-point computation, IEEE 754 defines "five types of exceptions that shall be signalled when detected" [IEEE85]. It allows for two types of signals: either setting an appropriate status flag or taking a trap. While it must be possible for the user to catch these traps, the default is to continue without trapping. The status flag must collect all exceptions that happened since the last reset of its contents; such a reset must be initiated by the user. This way of handling the status flags enables the user to control the program's behaviour even in the presence of exceptions. IEEE 754 defines the following five exceptions:

- *Invalid operation*: This exception is signalled if an operand is invalid for the operation to be performed. This includes the divisions $0/0$ and ∞/∞, the multiplication $0 \times \infty$, taking a square root of a number less than zero,

certain comparisons between unordered operands, magnitude subtraction of infinities, and a few more. For a complete list, see [IEEE85, p. 11].

When an invalid exception occurs, a quiet NaN is returned for operations with a floating-point target format.

- *Division by zero*: "If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception shall be signalled. The result, when no trap occurs, shall be a correctly signed ∞."

- *Overflow*: "The overflow exception shall be signalled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded." When no trap occurs, the result is determined by the rounding mode and the sign of the intermediate result. For trapped overflow, IEEE 754 aims to preserve as much information from the operation as possible by requiring the implementation to include as many sensible bits as possible. For details on the returned result, see [IEEE85, p. 11].

- *Underflow*: When a result in the subnormal range is returned, i. e. its absolute value is less than 2^L, in many cases the result is exact (cf. [Mul+10, p. 124]). However, in some circumstances, a significant loss of accuracy can occur. IEEE 754 defines two events which contribute to an inaccurate very small result: loss of accuracy and tininess (i. e. a subnormal number is returned). Tininess can be detected either before or after rounding, whereas loss of accuracy can be detected when the result differs from what would have been attained were either the exponent range unbounded or the exponent range and the precision unbounded.

  In [Cuy+02], the three cases are categorised as follows: For an arithmetic operation $x \mathbin{\circledcirc} y$ with $x, y \in \mathbb{F}^*(\beta, t, L, U)$, let r_e be the exact result (unbounded precision and unbounded exponent range), r_t the normalised result rounded to a precision of t digits (with unbounded exponent range), and $r \in \mathbb{F}^*(\beta, t, L, U)$ the returned result (which might be a subnormal number). Then,

  - *u-underflow* occurs when $|r_t| < \beta^L$ and $r \neq r_t$,
  - *v-underflow* occurs when $|r_t| < \beta^L$ and $r \neq r_e$, and
  - *w-underflow* occurs when $|r_e| < \beta^L$ and $r \neq r_e$.

  Note that u-underflow always implies v-underflow (since denormalisation loss implies inexactness), and v-underflow implies w-underflow (since tininess after rounding implies tininess before rounding).

  In absence of an underflow trap, the underflow exception is only signalled when both tininess and loss of accuracy occur; with trapped underflow, it is

signalled when tininess is detected, and similar to the overflow case, a result is returned that preserves as much information as possible (cf. [Mul+10]).

- *Inexact*: To cite IEEE 754 once more, "If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception shall be signalled. The rounded or overflowed result shall be delivered to the destination or, if an inexact trap occurs, to the trap handler."

It should be noted that of these exceptions, only overflow with inexact and underflow with inexact can occur at the same time.

If a trap handler is implemented to allow direct reaction to exceptional circumstances, the standard requires it to be a "subroutine-like function" that provides the user with information about the type of exception that occurred, the operation, as well as the destination format. For underflow, overflow, and inexact, the correctly rounded result, possibly including additional information (see above), needs to be returned; for invalid and divide by zero, the operand values must be supplied by the trap handler.

Miscellaneous

IEEE 754 concludes with Annex A that recommends a number of functions and predicates as "aids to program portability across different systems". These include functions to copy the sign of one operand to another operand or to generate the next representable floating-point number of an operand, classification functions to find out whether an operand is finite or an NaN, and additional comparison functions.

1.2.2 IEEE 854

IEEE 754 deals only with binary floating-point numbers, i. e. with base $\beta = 2$. However, most of the principles described in the standard can also be applied to arbitrary bases. Therefore, two years later a new standard called "IEEE Standard for Radix-Independent Floating-Point Arithmetic" was published as IEEE 854. It allows for bases $\beta = 2$ and $\beta = 10$ by generalising all definitions that formerly only applied to $\beta = 2$. Furthermore, a few clarifications were added.

The differences between both standards from a practical point of view can be best summarised by citing the foreword of IEEE 854: "The committee believes that, except for a possible conflict with the requirements in 5.6 and 7.2 that unrecognisable decimal input strings signal an exception, and in 6.3 that the sign of zero be preserved in certain conversion operations, any implementation conforming to ANSI/IEEE Std 754-1985 will also conform to this standard. In addition, the definition of `logb` has been enhanced in the Appendix, and two new functions, `conv` and `nearbyinteger`, have been added."

1.2.3 IEEE 754-2008

After publication of IEEE 754 and IEEE 854, the new standard was widely adopted. While not all designers of floating-point hardware and software implemented all features required by IEEE 854, e. g. providing only one rounding mode (usually round to nearest) or not supporting exceptions, the new standard raised the bar for floating-point implementations considerably and succeeded especially in providing standard formats, exceptions, and rounding modes, as well as requiring the basic operations to be rounded correctly. Virtually all floating-point arithmetic since IEEE 754 has been implemented adhering to its basic principles and incorporating most of its requirements, leading indeed to better portability, comprehensibility, and reliability of numerical algorithms.

Approaching the end of the last millennium, it was deemed necessary to revise the standard in order to merge IEEE 754 and IEEE 854, extend the old standards where necessary, and render some definitions and requirements more precise (e. g. conversion from binary to decimal representation). The revision process started in 2000 and culminated in the approval of the "IEEE Standard for Floating-Point Arithmetic", also called IEEE 754-2008, on 12 June 2008.

It is important to note that not only did the principal design goals (numerical robustness, reliability, and portability) remain the same, but that it was aimed to provide a standard that still accepted all platforms as conforming that conformed to IEEE 754 and IEEE 854, i. e. the new standard should not invalidate any hardware that conformed to its older versions [Mul$^+$10, p. 79].

That being said, the revision provided for the opportunity to reformulate the standard's principles in a more precise manner, while at the same time accounting for advances in floating-point design and the underlying numerical methods by requiring or recommending new operations and alternative exception handling facilities. Citing from its foreword, "This standard provides a discipline for performing floating-point computation that yields results independent of whether the processing is done in hardware, software, or a combination of the two. For operations specified in the normative part of this standard, numerical results and exceptions are uniquely determined by the values of the input data, the operation, and the destination, all under user control." The foreword ends by expressing hope "that language designers will look on the full set of operation, precision, and exception controls described here as a guide to providing the programmer with the ability to portably control expressions and exceptions. It is also hoped that designers will be guided by this standard to provide extensions in a completely portable way." [IEEE08]

IEEE 754-2008 consists of two parts: a *normative* part that conforming implementations must adhere to, and a *recommendatory* part that describes operations and facilities that provide for better numerical results and better portability. The standard notes what is needed for an implementation to be called conforming: A "programming environment" must support the required

operations, formats, and features of the normative part in either of the two supported bases $\beta = 2$ or $\beta = 10$. It goes on by listing some programming environment considerations: "Conformance to this standard is a property of a specific implementation of a specific programming environment, rather than of a language specification." [IEEE08]

Formats

IEEE 754-2008 distinguishes between basic, interchange, extended, and extendable formats. It defines three binary basic formats called binary32, binary64, and binary128, as well as two decimal basic formats (in binary representation, called decimal32 and decimal64). Interchange formats are fixed-width encodings that can be used for the exchange of floating-point data between implementations; they are identified by their size. An extended precision format extends a basic format by using a representation with both larger exponent range and a longer significand. For an extendable precision format, the range and length of the significand can be specified by a user.

A conforming implementation must at least implement one of the basic formats with means to initialise this format and convert values between that format and all other supported formats, and a corresponding interchange format with means to read and write that format. Furthermore, the required operations for the implemented basic format(s) must be supported.

It should be noted that IEEE 754-2008 does not define completely new formats. In fact, binary32 is IEEE 754's single format, and binary64 is IEEE 754's double format. Furthermore, Intel's x87 double-extended 80 bit format is called binary64 extended in IEEE 754-2008. While most programming languages do not offer extendable formats, additional floating-point libraries can support such formats, such as MPFR (see Section 5.7.3 and [GNU16d]). Table 1.1 lists the formats used in this thesis as well as their names. For historical reasons and the current naming conventions in programming languages, the "old" names such as single and double will be used instead of binary32 and binary64.

Format	Name	Width	Exponent	Significand	Exponent bias
binary16	half	16	5	11	$2^4 - 1 = 15$
binary32	single	32	8	24	$2^7 - 1 = 127$
binary64	double	64	11	53	$2^{10} - 1 = 1023$
binary128	quadruple	128	15	113	$2^{14} - 1 = 16382$
binary64 ext.	extended	80	15	64	$2^{14} - 1 = 16382$

Table 1.1: *(Binary) Floating-point formats. Partly taken from [WIK17q]. Width, exponent, and significand are given in bits; with the exception of the extended format, the significand includes the hidden bit.*

Attributes and Rounding

IEEE 754-2008 defines attributes that are "logically associated with a program block to modify its numerical and exception semantics. A user can specify a constant value for an attribute parameter" [IEEE08, p. 15]. In particular, these are rounding-direction attributes, alternate exception handling attributes (which both existed in the former standards, albeit without being called attributes), preferred width attributes, value-changing optimisation attributes, and reproducibility attributes. The standard mandates that rounding-direction attributes must be supported.

For binary floating-point arithmetic, the four rounding modes described in IEEE 754 remain in IEEE 754-2008 as they are. Additionally, when rounding to nearest, a new tie-breaking rule is added, leading to a new rounding mode *round to nearest, ties to away*: When an exact value lies exactly between two representable floating-point numbers, the one with larger magnitude is returned. This rounding mode is especially relevant for decimal floating-point arithmetic as this introduces the so-called "kaufmännische Rundung" [DIN92]. For binary implementations, it is not required. Table 1.2 lists the rounding modes and their respective attribute names which will be used in this thesis.

IEEE 754-2008 rounding-direction attribute	IEEE 754 rounding mode
roundTiesToEven	round to nearest
roundTiesToAway	—
roundTowardPositive	round toward $+\infty$
roundTowardNegative	round toward $-\infty$
roundTowardZero	round toward 0

Table 1.2: *Rounding modes and attributes.*

Operations

IEEE 754-2008 clarifies some of the operations listed in the former standards, especially concerning comparisons, and adds new required functions like `nextUp`, `nextDown`, `minNum`, and `maxNum`. Additionally, some formerly recommended functions like `copy`, `negate`, `abs`, or `copySign` are now mandatory. For decimal floating-point arithmetic, some specialised operations like `sameQuantum` and `quantize` are added, as well as preferred exponents for decimal results for all operations.

From an arithmetic point of view, the most notable change with regard to required operations is the incorporation of the `fusedMultiplyAdd` (`fma`) operation which has found its way into numerous processor designs as many scientific workloads require the multiplication of two values directly followed by an addition. IEEE 754-2008 mandates $\texttt{fma}(x, y, z) = (x \times y) + z$ to be calculated with unbounded range and precision, and rounded only once to the destination format.

Concerning conversion between binary floating-point numbers and decimal representations, IEEE 754-2008 requires correctly rounded base conversions for all possible values. This is a stricter requirement compared to the older standards (cf. page 11).

At the time when IEEE 754 and IEEE 854 were published, efficient algorithms and implementations for trigonometric, exponential, and logarithmic functions were not known, so these operations were not mentioned at all in these standards. Although at the time of the revision process enough knowledge and computing power was available to implement and compute correctly rounded versions of these operations in an efficient manner (cf. e. g. [Lau08]), they were only added to IEEE 754-2008 as recommended operations, including definitions for special cases.

Furthermore, IEEE 754-2008 recommends reduction operations that take a vector of operands in one format and return a result in the same format, thus reducing a vector of operands into a single result. These operations include e. g. `sum`, `dot`, `sumSquare`, `sumAbs`, or `scaledProd` and are recommended for all supported arithmetic formats.

Exceptions and infinity arithmetic

With regard to exceptions and arithmetic with special values such as infinities and NaNs, the definitions of the older standards were revised and clarified, but no substantial changes were introduced.

The clause about alternative exception handling was extended to allow optional exception handling in various forms, including traps and other models such as try/catch [WIK16].

Reliability and portability

To facilitate better reliability and portability of numerical programs, IEEE 754-2008 adds some clauses to address different facets of modern programming languages and compilers, such as expression evaluation rules, assignments, function values, and optimisations. It recommends guidelines and principles to handle these in portable manners. Furthermore, it adds optional `preferredWidth` attributes which can be used to change the format of intermediate computations for program blocks. This enables the user to increase the intermediate precision for accuracy-sensitive parts of an algorithm.

Furthermore, a clause was added dedicated to the reproducibility of floating-point computations. It notes that “reproducible results require cooperation from language standards, language processors, and users” [IEEE08, p. 51] and suggests means to control how floating-point operations are (reproducibly) performed.

Finally, IEEE 754-2008 concludes with Annex B about program debugging support, giving guidelines about which features are needed from a debugger to efficiently find numerical bugs in a floating-point program.

This overview of IEEE 754-2008 concentrated on the main differences to its earlier versions IEEE 754 and IEEE 854 as far as they are needed for this thesis. For an overview of the revision process and a more detailed analysis see [WIK16]; for detailed information about the entire standard, the reader is referred to the revision website [Hou08].

1.2.4 IEEE 754-2018

According to the rules of the IEEE, standards have a limited validity of 10 years. After this period, the standard must either be revised or withdrawn, otherwise it is assigned an inactive status (see [IEEE17b]). To maintain a valid active floating-point standard, IEEE 754-2008 needs to be revised by 2018. Therefore, mid-2015 "a minor revision has been undertaken, to clean up its errata and republish it mostly unchanged" [Hou17b]. During the first meeting, the scope of this minor revision was explicitly stated as follows: "The scope of this activity is to produce a timely correction to the 2008 standard, fixing errors and ambiguities and avoiding major restructuring, additions, deletions, changes in behaviour, or other controversial actions." [Hou15] It is intended to start a more thorough revision process in 2019 towards producing the next iteration (tentatively called IEEE 754-2028).

The list of errata of IEEE 754-2008 has been chosen as a starting point for the current revision [IEEE17a], with additional topics being raised during meetings. The most notable topics discussed for IEEE 754-2018 so far are unifying the use of the terms function, operation, and predicate, special cases for the `pown` operator[4], setting minimum values for small extendable formats (precision $t \geqslant 3$ and $U \geqslant 2$ for $\beta = 2$), and the addition of definitions for `asinPi` and `acosPi`[5] (which are missing in IEEE 754-2008). Suggestions deemed to be out of scope for the current process are postponed to IEEE 754-2028.

1.3 Rounding and error analysis

Executing numerical computations on any platform with limited precision arithmetic (such as floating-point arithmetic) inevitably leads to errors. [Hig02] mentions three main causes for these errors:

- Rounding errors. These are unavoidable due to the limited precision.
- Data uncertainty. This may stem from measurement or estimation errors when working on "real-world" data, from perturbations while storing or converting the data (which basically comes down to rounding errors), or from earlier computations performed on the original data.

[4] $\texttt{pown}(x, n) = x^n$ for $x \in \mathbb{R}$ and $n \in \mathbb{N}$.

[5] For $x \in [-1, 1]$, these are defined as $\texttt{asinPi}(x) = \arcsin(x)/\pi$ and $\texttt{acosPi}(x) = \arccos(x)/\pi$.

- Truncation (discretisation) errors. These errors are introduced by modelling the problem and choosing an algorithm to perform the calculation. For instance, many standard numerical methods can be derived by taking finitely many terms of a Taylor series. The number of terms used to compute the result directly influences the precision (and thus the error) and the speed of the computation.

In the context of this thesis, we only focus on the errors caused by rounding. In the following, we introduce basic terms and definitions related to error analysis and also shortly cover stability of algorithms. For a more detailed analysis, see chapters 1 and 2 of [Hig02] and chapter 2 of [Mul$^+$10] (where most of this section is based upon).

1.3.1 Some basic properties

Definition 1.1. Let $x \in \mathbb{R}$ and $\circ : \mathbb{R} \to \mathbb{F}^*(\beta, t, L, U)$ one of the five rounding functions required by IEEE 754-2008. Then the *absolute error* is given by

$$\epsilon_{\text{abs}}(x) = |x - \circ(x)| \tag{1.3}$$

and the *relative error* by

$$\epsilon(x) := \epsilon_{\text{rel}}(x) = \left| \frac{x - \circ(x)}{x} \right|. \tag{1.4}$$

$\Diamond$

For $x \in \mathbb{R}$ and $\circ(x)$ in the normalised range of $\mathbb{F}(\beta, t, L, U)$, it holds

$$\epsilon(x) < \beta^{1-t}, \tag{1.5}$$

whereas $\epsilon(x)$ can be as large as 1 if $\circ(x)$ is subnormal. In that case, the following bound holds:

$$\epsilon_{\text{abs}}(x) < \beta^{L-t+1}. \tag{1.6}$$

When round to nearest is used, these error bounds can be further improved:

$$\epsilon(x) \leqslant \frac{1}{2}\beta^{1-t} \tag{1.7}$$

and

$$\epsilon_{\text{abs}}(x) \leqslant \frac{1}{2}\beta^{L-t+1} \tag{1.8}$$

for $\circ(x)$ in the normal or subnormal range, respectively.

The above results can be generalised by combining the relative and absolute error bounds: If $z = \circ(z_1 \vartriangle z_2)$ for $z_1, z_2 \in \mathbb{F}(\beta, t, L, U)$ and no overflow occurs, then

$$z = (z_1 \vartriangle z_2)(1 + \delta) + \gamma \tag{1.9}$$

with $|\delta| < \beta^{1-t}$ and $|\gamma| < \beta^{L-t+1}$ (or, in case of round to nearest, $|\delta| \leqslant \frac{1}{2}\beta^{1-t}$ and $|\gamma| \leqslant \frac{1}{2}\beta^{L-t+1}$). Moreover, when z is subnormal, then $\delta = 0$, and when $z \in \mathbb{F}(\beta, t, L, U)$, then $\gamma = 0$ [Kah96b]. The bound on δ (i. e. $\delta = \beta^{1-t}$ and $\delta = \frac{1}{2}\beta^{1-t}$ for round to nearest) is also called unit roundoff, see Definition 1.4.

When analysing the quality of floating-point operations, it is more useful to evaluate the relative error introduced by an operation (due to the dependency of the precision relative to the floating-point number's exponent). However, when trying to assess if the result of a floating-point operation is correctly rounded, the following measure gives accurate insight into the operation's performance:

Definition 1.2. If $x \in \mathbb{R}$ with $x \in [\beta^e, \beta^{e+1})$, then

$$\operatorname{ulp}(x) = \beta^{\max(e,L)-t+1}. \tag{1.10}$$

ulp is also called *unit in the last place*. ◊

Note that when x is a floating-point number, the definition of ulp coincides with the quantum of x:

Definition 1.3. Every floating point number x can be represented as

$$x = S_x \times \hat{s}_x \times \beta^{e_x-t+1} \tag{1.11}$$

with S_x the sign of x, $\hat{s}_x$ the integer significand, and e_x the exponent. Then,

$$\beta^{e_x-t+1} \tag{1.12}$$

is called the *quantum* of x. ◊

Note that the definition in equation (1.11) is equivalent to that in equation (1.2) (with base β instead of base 2).

Another term closely related to ϵ and ulp that is widespread in the analysis of numerical algorithms is the term *unit roundoff*:

Definition 1.4. Given a floating-point system $\mathbb{F}(\beta, t, L, U)$, the *unit roundoff* $\mathbf{u}$ is defined as

$$\mathbf{u} = \begin{cases} \dfrac{1}{2}\operatorname{ulp}(1) = \dfrac{1}{2}\beta^{1-t} & \text{in round to nearest mode,} \\ \operatorname{ulp}(1) = \beta^{1-t} & \text{in the other rounding modes.} \end{cases} \tag{1.13}$$

The unit roundoff is also sometimes called *machine epsilon*. ◊

For any arithmetic operation $\triangle \in \{+, -, \times, /\}$, for any rounding mode $\circ$, and for all floating-point numbers z_1, z_2 such that $z_1 \triangle z_2$ does not underflow or overflow, we have

$$\circ(z_1 \triangle z_2) = (z_1 \triangle z_2)(1 + \epsilon_1) = \frac{(z_1 \triangle z_2)}{(1 + \epsilon_2)} \tag{1.14}$$

with $|\epsilon_1|, |\epsilon_2| \leqslant \mathbf{u}$.

1.3.2 Stability

To evaluate the correctness of a floating-point operation, i. e. to verify that the result of a floating-point operation is correctly rounded, it is sufficient to regard the error in terms of ulps: If the returned result is off by more than one ulp, it surely is not correctly rounded. Furthermore, even for operations that are difficult to round (such as elementary functions), it suffices to regard either the error in ulps or the relative error ϵ.

However, the situation is different when evaluating the quality of a result gained by executing a number of floating-point operations (in other words, the quality of the result of a numerical algorithm). When computing an approximation $\tilde{y}$ to $y = f(x)$ for $x, \tilde{y} \in \mathbb{F}(\beta, t, L, U)$ and $y \in \mathbb{R}$, the best possible error bound would be a small absolute error ϵ_{abs}, but in general, such an error bound will not be possible to achieve due to the limited precision of floating-point numbers. The next best possible result would be small bounds on the relative error ϵ, ideally with a magnitude of $\mathcal{O}(\mathbf{u})$, i. e. $\epsilon(\tilde{y}) \approx \mathbf{u}$. If such error bounds can be achieved, the algorithm is said to be *forward stable*, and the errors are called *forward errors*.

It is not always possible to achieve such error bounds. In that case, a different approach is helpful by asking "For what set of data have we actually solved our problem?" In other words, one tries to find a minimal perturbation Δx of the input x such that $\tilde{y} = f(x + \Delta x)$. If such a minimal $|\Delta x|$ or, depending on the situation, the relative variant $|\Delta x|/|x|$ exists, it is called *backward error*. An algorithm for computing $y = f(x)$ is called *backward stable* if, for any x, it produces a computed result $\tilde{y}$ with a small backward error, i. e., $\tilde{y} = f(x + \Delta x)$ for some small Δx.

A result of the form

$$\tilde{y} + \Delta y = f(x + \Delta x), \qquad |\Delta y| \leqslant \delta |y|, \quad |\Delta x| \leqslant \gamma |x| \tag{1.15}$$

is called a *mixed forward-backward error* result. In other words, $\tilde{y}$ is almost the right answer for almost the right data, provided that δ and γ are sufficiently small.

Finally, an algorithm is called *numerically stable* if it is stable in the mixed forward-backward error sense of equation (1.15). With this definition, a backward stable algorithm is numerically stable (set $\Delta y = 0$ in equation (1.15)).

1.3.3 Use cases for rounding

In this section, we briefly touch a few reasons why IEEE 754-2008 demands different rounding modes. Especially when considering equation (1.7) and (1.13), it is a legitimate question to ask why using round to nearest mode might not be the best choice.

Interval arithmetic

Interval arithmetic is "an approach to putting bounds on rounding errors and measurement errors in mathematical computation and thus developing numerical

methods that yield reliable results" [WIK17r]. This is achieved by bounding all variables into intervals consisting of upper and lower bounds. To ensure that all values are indeed kept inside these bounds, the rounding mode is switched to round towards $-\infty$ when calculating the lower bound and to round towards $+\infty$ for the upper bound. Although a higher relative error ϵ might be produced, the correct solution is guaranteed to stay inside the interval. When using round to nearest mode, slightly tighter bounds could be achieved, but the correct solution might be just outside the interval bounds. For more details on interval arithmetic, see e. g. [Moo66], [Kul89], or the recently adopted "IEEE Standard for Interval Arithmetic" (IEEE 1788-2015 [IEEE15], which relies on the underlying floating-point implementation to be conforming to IEEE 754-2008).

Analysing numerical stability

Using different rounding modes is also useful in diagnosing numerical stability: If the results of a subroutine do not vary substantially between variants that use roundTiesToEven, roundTowardNegative, roundTowardPositive, or roundTowardNegative, it is likely numerically stable. Also, if results vary substantially, the method might be unstable and affected by rounding errors (see also [WIK17m]). This reasoning is the basis for techniques like repeated randomised rounding (see e. g. CESTAC [Vig93], CADNA [LCJ10], and VERROU [FL16]) or MCA (Monte Carlo Arithmetic, see e. g. [Par97] or Verificarlo [DDP15]). These techniques randomly change the rounding mode during the runtime of the algorithm whose stability is to be analysed.

However, Kahan vehemently states that rounding errors are not uniformly distributed and that repeated randomised rounding and MCA sometimes destroy useful numerical properties such as the Exact Cancellation Theorem ([Kah98, p. 17], cf. also [Kah06]). He argues that randomised rounding can increase the confidence that a given method is numerically stable if the results vary only slightly between different runs, provided they are compared with known correct results. If on the other hand results vary significantly, Kahan strongly advises to employ other means of analysis to verify the numerical stability of the algorithm

Numerical algorithms

Finally, we mention two examples of numerical algorithms that rely on being able to switch the rounding mode. [Kor+12] proves that for $a, b, c \in \mathbb{F}(\beta, t, L, U)$ and $RN : \mathbb{R} \to \mathbb{F}^*(\beta, t, L, U)$ the roundTiesToEven rounding function, no algorithm exists that calculates $RN(a + b + c)$ using only computations in roundTiesToEven mode. Instead, they present an algorithm relying on round-to-odd addition (cf. [BM08]) which uses roundTowardPositive and roundTowardNegative.

In [Rum13b], a variant of an algorithm that uses extra-precise accumulation of dot products to solve ill-conditioned linear systems accurately is presented (cf.

[Rum13a]). The results of that algorithm are significantly improved by permitting directed rounding modes.

1.4 The user environment

This thesis is about IEEE 754-2008 conformity regarding platforms as well as IEEE 754-2008 support of the components necessary to execute (numerical) programs on that platform. Running numerical algorithms involves a lot more components than visible at first glance, amongst which are the programming language of choice, the compiler, and the hardware on which the program is executed. Ideally, it should not be necessary for a user to fully understand all parts involved, but rely on the means provided by the chosen programming language and trust that all other components work together following the philosophy of the standard, or more precisely, that all components adhere to the standard as far as needed to supply a conforming platform as a whole.

With this in mind, IEEE 754-2008 explicitly specifies that it "provides a discipline for performing floating-point computation that yields results independent of whether the processing is done in hardware, software, or a combination of the two" [IEEE08, p. iv], and it states that "conformance to this standard is a property of a specific implementation of a specific programming environment, rather than of a language specification" [IEEE08, p. 2]. This means conformance is regarded as a property of a platform in its entirety.

A platform cannot be conforming to the standard if at least one component does not support all relevant parts of the standard: Consider for instance processor hardware that supports the required rounding modes, but does not signal exceptions. In this case, it is not possible[6] to gain insight into (numerical) problems encountered inside an algorithm, e. g. it might not be possible to understand why a subroutine returned an NaN. Of course, such a platform cannot be considered fully IEEE-compliant.

Note though that if parts of the execution chain are known to be incompatible, it might be possible to make the platform conforming by replacing the computations done on non-conforming components with adequate software implementations. Furthermore, only those parts of the involved components which are actually exercised when running a specific numerical program need to be IEEE-conforming.

What we call the *user environment* comprises a platform in its entirety: A set of computer hardware, operating system, numerical and system libraries, a programming language and the respective compiler, and even the compiler switches used. In the following, we examine the components involved in executing programs incorporating floating-point calculations, and in which part of hardware or software floating-point operations are executed. This directly influences reproducible and

[6] In theory, it is possible to emulate or recover exceptions purely in software; in reality however, this is far too costly in terms of performance.

reliable execution of floating-point programs and the IEEE-conformity of a given user environment or the parts it consists of.

Before describing the components of a floating-point environment in detail, we define the term user environment:

Definition 1.5. A (floating-point) *user environment* consists of everything that influences how a floating-point program is executed. More specifically, the following components constitute a user environment:

- the computer hardware, consisting of a CPU and optional FPUs or accelerators,
- the operating system enabling programs to execute on that hardware, including mathematical libraries that come with the operating system,
- the programming language in which the user program is written,
- the compiler that is used to compile source code into machine instructions, including all compiler options set during compilation,
- mathematical libraries supplied by the compiler,
- optionally an interpreter needed to execute code,
- and additional mathematical libraries linked into the program.

Such an environment in its entirety is also called *floating-point environment* (or simply *environment* if the context is clear) in this thesis.

Note that we use the term *platform* in this thesis when we refer to the underlying computing hardware of a user environment. ◊

This definition allows for a broad range of fundamentally different floating-point environments: software packages that perform all floating-point computations in software; "normal" office computers that use some kind of spreadsheet program (which might use the CPU's floating-point or multimedia unit to execute floating-point operations); specialised supercomputers used for large-scale scientific simulations that can schedule floating-point calculations to different execution units, e. g. vector extensions or accelerators such as GPUs or FPGAs; or even virtual machines running inside a virtualised computing environment (consider e. g. cloud computing). While it is not feasible to assess the quality of the floating-point operations on every floating-point environment,[7] this definition allows for a standardised way of analysing the components involved in executing floating-point operations.

[7] This is valid especially for floating-point software packages as it might be difficult or even impossible to interface a software package from an external testing program.

1.4.1 Limitations of the user environment definition

Definition 1.5 takes a rather CPU-centric point of view: The user environment is considered to be a set of one piece of computer hardware with some CPU, an operating system, and further software that is needed to execute numerical algorithms. It also allows for the addition of some FPUs or (probably internal) accelerators (cf. Section 1.4.2) which are in most cases accessed by using special library commands. However, it does not account for the most common model in High Performance Computing (HPC) systems where programs are executed on a (possibly very large) set of identical compute nodes, i. e. when the user environment does not consist of one but rather several computers.

This mode of execution poses two main challenges for tools that analyse the floating-point conformity of such an environment: If the IEEE-conformity of this whole floating-point environment (or rather set of environments) is to be evaluated, every computing node must be evaluated on its own, thus ensuring that the nodes behave identically when given the same (numerical) computing task. Some of the challenges that arise are discussed in Sections 1.4.3 and 3.1.4.

Additionally, parallel programs need to exchange data between different compute nodes, thereby employing another set of libraries handling these communications. It is reasonably safe to assume that copying floating-point numbers from one compute node to the next does not change the values of these numbers. However, to speed up communication between nodes and to relieve the CPU of some of its compute burden, vendors start employing an offload approach that shifts computation of certain operations from the compute node to the network switch. For a detailed discussion of network offloading, in-network computations, and the impacts on IEEE-conformity, see Section 1.4.8.

1.4.2 Floating-point hardware

Although a fully IEEE-conforming floating-point environment can be written in software regardless of the underlying hardware, it is highly desirable, especially from a performance point of view, that the hardware provides means to execute floating-point operations in an efficient manner. The more IEEE 754-2008 features are supported in hardware, the less work has to be emulated in software which would significantly slow down numerical programs.

In the beginnings of floating-point computations in the 1960s, the only execution unit available on computers of that era was the CPU which typically consisted of a few registers to store data, commands to control the program flow and move the data, and commands to process the data, including instructions to perform integer operations. Floating-point operations had to be written in software by utilising the available commands, especially those of the integer unit. If a floating-point standard had existed at that time, it would have been the emulating software's responsibility to ensure conformance to that hypothetical

standard.

Since then, processor hardware designs have become much more complex, providing comprehensive instruction sets, specialised processing units for different needs (like floating-point instructions or cryptographic operations), and complex memory hierarchies. Virtually all modern processors targeted either directly for scientific computing or for all-purpose computing include some type of floating-point unit (FPU) specifically designed to perform floating-point operations. As a consequence, of all floating-point operations available to the user, those for which an implementation exists either in the CPU's instruction set or in an FPU can be executed in hardware, which usually results in faster performance.

FPUs

Since doing floating-point computations purely in software is very costly and therefore slow, designers of computer hardware started developing dedicated floating-point units that execute floating-point instructions in hardware, thereby massively reducing the execution time of basic floating-point calculations. Although already available in the 60s in mainframes such as the IBM System/360, widespread adoption of such specialised floating-point units even in consumer hardware started only in the 80s with the introduction of Intel's 8087 floating-point coprocessor (which was a separate piece of hardware complementing the 8086 processor). The 8087 consisted of floating-point registers to hold floating-point values, an instruction set, and a status register that contained information about the currently selected rounding mode and all exceptions that occurred since the last reset of the register. The instruction set contained instructions to transfer floating-point data between main memory and the floating-point registers, mathematical operators such as addition or square root, comparisons between floating-point numbers, conversions between floating-point and integer values, and instructions to operate on the status register like reading and setting values. For a more detailed description, see Section 5.2.1.

Already one of the next generations of Intel CPUs incorporated the then-called 487 FPU extension into the 486 design and kept it on the same processor die. This is highly advantageous from a memory transfer point of view because the CPU gains direct access to the floating-point registers. Since then, the majority of processor designs unite some type of floating-point execution unit and special floating-point registers together with all-purpose execution units on the same processor die to support fast-performing numerical calculations. Examples of FPU units are Intel's SSE and AVX units, ARM's VFP extensions, or the AltiVec and VSX instructions in POWER processors.

The design of these FPUs however has advanced to reflect current needs: Vectorised FPUs support performing floating-point operations on a vector of floating-point values at the same time, and deep pipelines increase FPU throughput.

(External) Coprocessors/Accelerators

Although specialised (external) numerical coprocessors like the x87 are not common any more, a new class of coprocessors which at least partially support floating-point operations has surfaced: accelerators like GPUs or FPGAs. GPUs (Graphics Processing Units), which were originally designed to speed up graphics calculations, have gained much interest as floating-point units due to their excellent price/performance ratio. They have become widespread under the term GPGPUs (General Purpose GPUs) after they started supporting IEEE-conforming floating-point calculations.

FPGAs (Field Programmable Gate Arrays) are processing units which can be reprogrammed to support arbitrary tasks. Due to their programmability, they are deployed in industry for a wide range of tasks such as audio DSP (Digital Signal Processing), real-time video engines, ASIC prototyping, industrial imaging, network switching and routing, hardware security modules, or data mining systems (for more applications and a coarse overview, cf. [WIK17l]). While possible in theory, it is not common to use an FPGA as a dedicated FPU. This is due to the fact that an IEEE-conforming floating-point implementation takes up a considerable amount of the logic gates (and registers) available on an FPGA, and the resulting performance is at best in the order of current FPUs.

Usually, execution of floating-point operations on these types of accelerators needs to be explicitly enabled, e. g. by using special compiler switches or linking corresponding libraries. This means that the user can control where floating-point calculations are performed.

1.4.3 The operating system

The operating system and its system libraries play a vital role as the link between user software and the actual hardware that is used to execute the software. It provides programs and compilers with programming interfaces to enable portability without recompilations and serves as an abstraction layer between the actual hardware and these programming interfaces. As a consequence, this means that the exact execution location of a floating-point operation might be hidden from the user, and it might not always be possible to determine in advance where the operation is performed:

- Depending on the actual hardware, especially the CPU model and available FPUs, the operating system might schedule floating-point instruction calls differently between systems. This is done either for performance reasons or to avoid known CPU bugs. One prominent occurrence of such a known error with different code paths depending on the actual CPU model is the infamous Intel Pentium 90 FDIV bug [INT04; Nic11; Ede97]: Most operating systems diverted the FDIV call to a slow, but correct software routine when

an affected CPU was encountered, whereas on "clean" Pentiums, the FDIV instruction implemented in the x87 FPU was called.

On Unix systems, such calls and routines are collected in the `libc`, such as in `glibc` on Linux systems. This approach has the advantage of transparently hiding bugs and other hardware quirks from the user. On the other hand, it might be difficult to trace which parts of a numerical algorithm are executed in software or in hardware (or even which part of the hardware).

Also, this approach might hide the fact that although the underlying CPU or FPU is IEEE-conforming, the software routines which are actually executed in favour of their hardware equivalents might be not.

- All relevant modern operating systems employ multitasking, i. e. the available computing time is split between different tasks. As a consequence, any task is assigned time slices in which it is executed. When the time slice is up, the current task is stopped, and execution is transferred to a different task. To guarantee that the stopped task is resumed exactly where it left off, the full task state needs to be conserved between context switches. This might be extremely expensive e. g. on RISC architectures with large amounts of registers. Thus, depending on the operating system and the platform's architecture, a tradeoff might be used to only save "important" registers and values.

  Consequently, whether and which registers related to floating-point operations are preserved between context switches depends on the operating system and the CPU of a platform and might be difficult to determine in advance.

- The same applies to multiprocessing and multithreading: As context switches might result in resuming the current task on a different core or even processor, full program information must be preserved between these context switches.

  This problem is especially severe on CPUs which use SMT (Simultaneous Multi-Threading): To perform context switches as efficiently as possible, usually only the minimally needed amount of registers is saved. Whether floating-point registers are affected depends again on the CPU and the operating system.

- Although on most modern operating systems special care is taken that context switches indeed preserve all relevant data, there is another consequence from multiprocessing and multithreading: Without taking extra measures like explicitly specifying processor/core affinity (if supported by the operating system), it is not possible to know in advance which core on which processor will be used to execute the program or whether the same core is used throughout the runtime of the program.

In most cases, this should not lead to severe consequences as processor cores on a given platform are usually identical; on the other hand, a faulty core could lead to completely non-deterministic behaviour of the program, sometimes yielding correct (and reproducible) results, sometimes yielding wrong values. That being said, faulty hardware always has adverse consequences for program correctness and reliability.

1.4.4 Floating-point libraries

In addition to the operating system's libraries, there are several other types of libraries which need to be considered regarding floating-point execution.

- Libraries needed to execute code on special hardware such as accelerators were already mentioned above. This type of libraries only serves as a gateway for the hardware execution units, and when targeted, it is clear that floating-point operations are executed on the hardware.

- System libraries have been shortly covered above. The most important system library with regard to floating-point calculations is the system's math library which includes calls for all types of mathematical operators like trigonometric, exponential and logarithmic functions, as well as power, remainder, and rounding functions. In most cases, calls to operators that are also implemented in an available FPU are directly passed to that unit, while operators which are not available in hardware are executed in software.

  There are two caveats: If a platform supports more than one hardware FPU, is is not clear which of those will be chosen to perform the calculation. Depending on the math library implementation, the best FPU unit might not be used although it exists in hardware. But even if the math library includes calls to all FPUs on the platform, there might not be a "best" choice: Consider e. g. a recent Intel Xeon server CPU which typically features an x87 FPU as well as SSE and AVX vector units. Favouring AVX over SSE should be the best choice between these two in terms of fastest code path, register usage, and execution time, but choosing the x87 FPU might yield better accuracy since all intermediate calculations are performed in extended format.

- Some compilers implement mathematical operators in their own math library. For details, see below.

- For many numerical algorithms, fast implementations exist for a wide range of platforms, such as BLAS [BLAS17], LAPACK [LAP17], MKL [INT17c], ARM Performance Libraries [ARMb], IBM ESSL an PESSL [IBM03a; IBM03b], Eigen [G+10], and many more (see also [WIK17t]).

- As an alternative to the floating-point formats used in programming languages (which usually directly map to hardware formats, see below), several software libraries with fully IEEE-conforming implementations of floating-point formats, operations, and attributes have been written. Using these libraries, perfect portability, reproducibility, and IEEE-conformity can be reached. Furthermore, some of these libraries offer fully customisable floating-point formats and precisions for intermediate calculations, resulting in correctly rounded results for all operations. From a performance perspective, these libraries are usually less than ideal due to management overhead and the library not being able to exercise special floating-point hardware.

  The most notable floating-point libraries in this context are MPFR [GNU16d], Berkeley SoftFloat [Hau17], and CRlibm [Dar+06], all of which are covered in detail in Chapter 5.

1.4.5 Programming languages

The programming language and its features are the main interaction point for a user with a given user environment. In general, it is not trivial to choose the best programming language for a given project, and this also holds when the desired goal is to implement numerical algorithms. The choice heavily depends on personal preferences as well as general considerations. If best possible performance is important, interpreted languages like `Java` or script languages like `Perl` or `Python` are at an disadvantage (for the the latter, this might not be true if special libraries like numPy [Num17] are used). Additionally, not all environments support compilers for all programming languages, so the choice may be limited. This is especially true for supercomputers like the Blue Gene/Q installation JUQUEEN at the Jülich Supercomputing Center [JSC17a] which might only offer `C`, `C++` or `FORTRAN` as possible programming languages.

From a floating-point point of view, the programming language influences how conforming a floating-point environment in its entirety can be since it provides the mapping between features available by the environment (be it hardware, libraries etc.) and the operations provided by the language. The following areas are of special concern:

- Supported formats: As stated above, IEEE-conforming environments must support at least one of the basic formats. Typically, all modern programming languages provide support for the single and double formats. Some languages such as `C` or `C++` also support an extended format called **`long double`**. Unfortunately, the exact format of **`long double`** sometimes depends on the underlying hardware platform: On Intel x86 processors, usually Intel's extended 80 bit format is used, whereas on SUN SPARCs **`long double`** maps to quadruple precision (emulated in software).

- Intermediate formats: Of special interest is the language's choice of intermediate formats: The standard recommends calculating intermediate results with higher precision and rounding towards the target format only at the end. For instance, when evaluating an expression with only single precision operands, all intermediate calculations should be performed in double precision. Common combinations are single/double, double/extended or double/quadruple.

  `Java` only supports single and double numbers, resulting in double expressions always being evaluated in double format. Furthermore, expressions involving only singles are also always evaluated in single, although it would be possible to use double as an intermediate format. This (among other things) led W. Kahan and J. Darcy to give the talk "How JAVA's Floating-Point Hurts Everyone Everywhere" [KD98] at the 1998 ACM "Workshop on Java for High-Performance Network Computing" about why this is a bad idea from a numerical point of view.

- Rounding modes: Choosing a rounding mode is usually done by performing a respective function call. This function (or family of functions) is sometimes supplied by the programming language (e. g. in `C99` [C99]) and sometimes by the compiler (e. g. `g++` for `C++`).

- For exceptions, the same as for rounding modes applies: Support is usually supplied by a family of functions either directly by the programming language or by the compiler/system libraries. This of course assumes that the underlying hardware supports exceptions.

- Available operators: Not all language standards supply mappings for all operators listed in IEEE 754-2008. However, in reality this does not limit the available range of operators since these can be supplied either as an extension by the compiler or through libraries.

1.4.6 The compiler's role

Another vital component when writing and executing floating-point algorithms on a given user environment is the compiler. Its role is particularly important as the compiler translates the means defined by the language into actual function or hardware calls and provides the mapping between language statements and execution unit. This is especially important on systems which provide more than one FPU or where more than one numerical support library is present. Also, the default execution targets for floating-point calculations might change between compiler releases. In the following, we give an (incomplete) enumeration of areas of concern:

- Evaluation of constants: Most programming languages provide means to define constants as mathematical expressions involving floating-point operations, which are converted into floating-point constants at compile time. As an example, a floating-point constant `twopi` with a value of $2 \cdot \pi$ might be defined in `C` as follows: `const twopi = 2 * M_PI`. Other languages even provide means to perform full floating-point algorithms at compile time, like `C++` via template meta-programming. How these calculations are performed is completely compiler dependent, and in general it should be checked that results are indeed calculated (and rounded) correctly. Often, a software floating-point library is used for this task (e. g. GNU `gcc` uses MPFR).

- The same applies for input and output of floating-point numbers (as decimal human-readable values): The routines used for these conversions are either directly supplied by the compiler, or a floating-point library providing the necessary means (like MPFR) is employed. The correctness of the necessary conversions is then completely dependent on these libraries, as well as on whether means to choose the rounding mode or to retrieve exceptions are available.

- For operations not supplied by the currently available hardware execution units on a given user environment, most compilers either use math libraries supplied by the operating system or implement their own (possibly higher performing) math libraries. Which of these libraries is chosen is compiler dependent.

Of course, compilers do not have to be regarded as black boxes which magically (and obliquely) choose which libraries or FPUs are employed to perform floating-point operations. Rather, they provide means to influence this choice, usually via compiler switches. Given these switches, completely differently behaving programs can be compiled out of the same source code. In the following, we discuss the most influential areas concerning compiler switches:

- The most obvious difference achieved by employing compiler switches was already mentioned: The FPU targeted for execution as well as the math library linked can usually be chosen via respective options.

  Sometimes, explicitly choosing the FPU is the only way of achieving consistent portable behaviour between different compiler versions: GNU `gcc` up to version 4.7 chooses SSE units as default FPUs on x86 platforms even on platforms supporting AVX, whereas from version 4.8 on, AVX is targeted by default.

- Some compilers provide compiler switches to turn off certain aspects of IEEE-conformity, e. g. ignoring exceptions or turning off subnormal numbers. By employing these compiler switches, faster performing code can be

generated or faster performing versions of the underlying algorithms (e. g. for multiplying floating-point numbers) can be selected which usually trade accuracy against execution time.

- If several sets of registers are available in the CPU, e. g. general purpose registers as well as special floating-point registers, compiler switches might be available to specify which registers are used for floating-point calculations and intermediate results. One well-known example is the switch `-ffloat-store` which `gcc` uses for x86 targets: If the x87 FPU is used in conjunction with this switch, all intermediate results are stored in the CPU's general purpose registers which are 64 bits long, effectively forcing rounding to double precision. Otherwise, the special x87 floating-point registers with extended precision can be used for intermediate results.

- The by far most influential compiler switches concerning floating-point calculations are optimisation switches: All modern compilers feature routines to optimise code for specific platforms in order to enable maximum performance. Unfortunately, many features necessary for IEEE-conforming calculations either prevent certain optimisations or perform tasks which are rarely needed, but deteriorate performance. When enabling aggressive optimisations, many floating-point relevant features are turned off, therefore reducing IEEE-conformity of the whole user environment. This especially applies to the evaluation of exceptions which are often dropped on higher optimisation levels. Additionally, the support of subnormal numbers might be turned off in some floating-point environments for performance reasons, e. g. on GPUs or in some Intel processors where floating-point computations involving subnormal numbers might be some orders of magnitude slower than calculations without subnormal numbers (see e. g. [HE02; MG14]).

  Furthermore, some optimisations might not affect conformity directly, but might alter floating-point semantics (and thereby affect portability):

  - Order of evaluation: Optimisers try to rearrange the order of execution of operations in an algebraically equivalent manner. However, as the associative law does not hold in general for floating-point numbers, these kind of optimisations might lead to different results compared to the same program compiled without optimisation.
  - Combining operations: When a fused multiply-add operation (`fma`) is natively available in the FPU, the compiler might combine multiplications and additions into fused multiplications and additions, thereby increasing performance. Additionally, numerical accuracy is usually better due to having to perform one less rounding per `fma`. On the other hand, results will be different compared to a version without `fma`.

Lastly, it should be noted that we slightly over-simplified the compilation process by ignoring the linking stage which also influences which math libraries or execution units are chosen. This is of some importance since, although the linker is a vital part of the build chain, it is usually part of the operating system and not of the compiler. However, the choice of libraries and execution units is usually done by the compiler, which then directs how the linker works by supplying the libraries to be linked via linker options, thus we feel this step can be ignored in this context.

1.4.7 Interpreters

Most programming languages are compiled to machine code and then executed directly on the target hardware. However, some languages introduce an intermediate step that either eliminates the need for explicit compilation (e. g. `Python`, `Perl`, `Matlab`) or improves portability by compiling to an intermediate format that can be executed on different hardware architectures via an interpreter, such as `Java`. In this case, the hardware abstraction and translation between hardware and language features is the responsibility of the interpreter.

The following models can be distinguished:

- Precompilation to an intermediate format. In this model, source code is compiled as usual. However, the compiler does not emit machine instructions, but intermediate code targeted at an interpreter (sometimes called virtual machine). When executing a program, its precompiled code in intermediate representation is then translated to machine instructions during runtime.

  The most prominent example for such a language is `Java`.

- Transparent compilation to an intermediate format during runtime. Source code can be executed directly via the language's interpreter, but the interpreter compiles whole source files into intermediate format before executing the resulting code.

  This is the standard compilation model of `Python` and `Matlab`.

- "On the fly" translation, i. e. source code is executed directly via an interpreter, and whenever an instruction is encountered (usually after parsing the whole file), it is translated into machine instructions and then executed.

  Most scripting languages (not all of which are arguably programming languages) employ this model, e. g. `Perl` and `Javascript`.

In the latter two cases, mixed models exist: While e. g. `Python` source code is usually compiled to a binary intermediate format during runtime, it is also possible to precompile modules to machine code and call these from "normal" `Python` code.

It is clear that aside from general language and compiler concerns (see above) the interpreter determines to what extent the user environment conforms to IEEE 754-2008 since only features offered by the interpreter are available to the user. This is different to other programming models where additional functionality can be added via external library calls. As an example, the `C++` and `Java` standards do not offer the possibility to set different rounding modes or to retrieve information on exceptions. However, in the `C++` case, this functionality is added by library calls that access the relevant CPU or FPU registers, whereas in `Java`, there is no possibility to extend a user program with these facilities.

Furthermore, how features provided by the language are mapped to actual hardware calls completely depends on the interpreter and its implementation for a given hardware architecture. Due to this reason, `Java` implements two base classes for floating-point operations, one for fast operations which may not yield identical results between different environments, and one that is aimed at portability and that guarantees that results are always identical regardless of the underlying hardware (but which may be slower). For a detailed analysis of `Java`'s floating-point capabilities and in how far the mapping between virtual machine and actual processor hardware influences IEEE-conformity, see Sections 5.7.1 and 6.7.3.

1.4.8 In-network floating-point computations

In their quest to simulate nature in the form of e. g. physical, chemical, or biological processes, scientists need to solve models of ever-increasing complexity and size, therefore creating demand for huge amounts of memory and computing power. This need can only be met by supercomputers consisting of several hundreds or thousands of compute nodes connected via high-speed networks. As a consequence, a typical scientific simulation will be executed as a single program running simultaneously on a large number of compute nodes, exchanging data via the high-speed interconnect.

Communicating data over any network involves some processing power to prepare the data according to the corresponding network protocol, sending the data, receiving it, and verifying its integrity. To alleviate the burden on the CPU, modern network implementations offload the handling of the communication protocol to the network interface card, thereby freeing valuable CPU cycles. For most communication operations such as copying data from one node to the other, this approach yields massive performance improvements. For certain global operations such as reductions (where values from all nodes or a subset need to be combined into one result, e. g. by summing all values) however, the values involved must be passed to the CPU, summed there, and then passed back into the network. To speed up this type of communication operation, vendors have started to offload the numerical part of the operation into the network, i. e. either the network interface card or the network switches (cf. e. g. [Mel17] where the

actual floating-point computations are performed inside the network switches).

While this approach is clearly desirable from a performance point of view, a number of questions arise with regard to the quality of the floating-point operations. For instance, which floating-point formats are supported fully depends on the network switch (or interface card) and the communication protocol/library, as well as the choice of available floating-point operations or the support of overflow and underflow, exceptions, or NaNs. Therefore, in order to assess the IEEE-conformity of such a supercomputer, not only the floating-point quality of the compute nodes needs to be evaluated, but it is vital to also check the computations done in the switch for IEEE-conformity. In the worst case, it might be necessary to avoid the reductions offered by the communication protocol and implement a custom reduction operation (which would be orders of magnitude slower, but known to be IEEE-conforming).

1.4.9 Resilience

As supercomputers have reached an execution speed of $\mathcal{O}(10)$ Petaflops and scientists are striving to develop an exascale computer, resilience has become a research topic of special interest. [Sni+14] This is due to the large scale of the existing and expected machines: The more components (i. e. nodes) are involved in the execution of a compute job, the higher the probability of at least one of the components failing during run time. Additionally, it becomes more likely that errors happen which either are not detected due to missing detection capabilities or because there exist no technical means to detect them at all. If a case of this so-called SDC (silent data corruption) happens in critical parts of the data, the numerical algorithm will be impacted, and the results might be rendered meaningless.

There are quite a number of approaches dealing with data corruption on an algorithmic level (e. g. [BS08; EHM14; Bou+15]) as well as tools to detect (and correct, if possible) errors introduced while transporting data (e. g. [Fia+12; NG13]) and to provide estimates how often it is feasible to checkpoint the current program state in order to enable rollback possibilities in case of errors, see e. g. [Dal06; Moo+10].

From an IEEE 754 point of view, not too much can be done with respect to resilience. When an FPU fails only occasionally or produces incorrect results only for certain (rare) combinations of operators and operands, it is almost impossible to detect these failures during run time, especially if the results are off only by a small number of ulps. If algorithms provide plausibility checks, it is at least possible to warn the user that parts of the computation might have produced wrong results and that further checks need to be executed.

In this context, a testing tool like `IeeeCC754++` can be of invaluable help as it might enable detection of such faulty CPUs or FPUs. For trivial cases, e. g. when every value that is returned by a FPU is wrong, a tool like `IeeeCC754++` might

not be necessary as the failures can be detected by simpler means. However, for more subtle cases, `IeeeCC754++` might able to detect errors which might otherwise go unnoticed. Extensions aimed at this use case are described in Section 3.1.4.

1.4.10 Comparing and testing floating-point environments

The former paragraphs illustrate clearly how complex a modern floating-point environment can be and emphasise the need to test the current environment from a user's point of view as well as the individual components that the user environment consists of. In this thesis, we will show how both of these aspects can be approached: by providing a flexible test tool that can either run in a generic mode (whose results are heavily dependent on the compiler and compiler switches set during building the application) or employ ports targeted at specific hardware FPUs or software libraries (see Section 3.3.3 and Chapter 5).

However, the discussion also shows that not all parts of the execution chain for performing floating-point calculations can be controlled by the user. Some of these concerns can be covered by testing the floating-point environment as a whole, but aspects such as data corruption or potential problems due to e. g. multithreading or processor affinity are almost impossible to detect and therefore hard to check.

Chapter 2

IeeeCC754

This chapter deals with the foundations of `IeeeCC754++` and particularly with its ancestry. Most of the material presented here is based on [VCV01a] and [VCV01b].

2.1 History

As with all complex hardware or software conforming to a given standard (or only to a promised feature set), it is an extremely hard problem to prove the correctness of a given implementation. To prove in a mathematical sense full compliance to the IEEE 754 standard, one would need to prove that every single floating-point operation for every possible combination of floating-point input operands yields the correct result. Although this approach has in fact been proposed for single precision operations on a single operand (e. g. in [Daw14] for `ceil`[1]), it is still not possible for operations with two or three operands, especially as far as the double or quadruple formats are concerned. The reason is simply combinatorial: Whereas there are $2^{32} \approx 4.3$ billion single floating-point numbers and thus 2^{32} `ceil` operations, which can be executed in a few seconds on a modern computer, there are $2^{64} \approx 1.8 \times 10^{19}$ distinct double floating-point numbers and thus 2^{64} `ceil` operations, which even under ideal circumstances would take more than 8 years to execute on a single processor, see the following example:

Example 2.1. Assuming a processor with 2 GHz clock frequency and only regarding the pure execution of the floating-point operation — neglecting data transfer from main memory to registers and the time needed to compare the results to known correct results — executing the `ceil` function for all possible single operands would take about 2 seconds on a single processor with only one floating-point execution unit (further assuming that one `ceil` execution per clock cycle can be

[1] The `C` library function `double ceil(double x)` returns the smallest integer value greater than or equal to `x`.

performed). Even if we assume a processor with 8 cores and 4 double floating-point execution units per core, executing `ceil` for all double numbers would take $2^{27} \cdot 2 = 2^{28} = 268,425,456$ seconds, which is about 3,107 days or 8.5 years. ◊

Still, it is highly desirable to check a given floating-point implementation for correctness and/or compliance to IEEE 754. In hardware development, a range of formal verification techniques are employed to verify the circuit layout before committing it to silicon [MLK98; Rus98; Rus99; Rus00; Har00a; Har00b]. Unfortunately, depending on the thoroughness of the verification model used, this process does not necessarily lead to correctly working processors (or, more precisely, floating-point units), as the infamous Pentium 90 division bug demonstrates [INT04].

Especially from the perspective of a computer user writing numerical code using floating-point numbers, it is a good principle not to simply assume (or trust) that the current user environment she or he is developing for conforms to IEEE 754. Therefore, it is important to have test tools available to check the given floating-point implementation (or only certain aspects of the implementation, e. g. the used underflow mode).

Furthermore, as described in detail in Section 1.4, the floating-point environment the user has access to is a complicated mixture of interacting software and hardware components consisting of the processor (possibly with some additional floating-point units), mathematical libraries, the operating system (which might call mathematical libraries), and the compiler. As not all parts of the compilation and execution chain are visible or even accessible by the end user, a floating-point environment is often experienced more or less as a "black box" in the sense that the user writes a numerical program in the programming language of choice, uses a compiler that is available (and maybe recommended) for the target computer, and simply executes the resulting program on that computer without further thinking of how and where exactly the floating-point operations used inside the program are executed.

Since the beginning of the 1980s, a number of tools have been developed to determine the basic characteristics of a floating-point environment and assess its IEEE-conformity. We briefly cover the tools which `IeeeCC754` is based upon, namely Paranoia, UCBTest, and a test suite developed by J.T. Coonen, while referring to [VCV01a, Chapter 2] for a closer look at these tools. We also refer to [Mul$^+$10, Chapter 3.8] where a few other floating-point related test tools are listed which are less relevant for this work.

Originally developed by W. Kahan, Paranoia [Kar85] was one of the first programs to assess the quality of a floating-point implementation. It determines basic properties of a given floating-point implementation, such as its precision, exponent range, supported underflow and rounding modes, etc. For a detailed example of Paranoia at work, see [Mul$^+$10, pp. 111-115].

Paranoia has been integrated into UCBTest [Hou$^+$88], which presents a "whole

set of programs for testing certain difficult cases of IEEE floating-point arithmetic" [VCV01a]. Additional programs included in UCBTest "generate difficult test cases for multiplication, division, and square root, respectively" [VCV01a]. They are chosen in a way that they are "difficult" to round in the sense that the exact results lie halfway or almost halfway between two floating-point numbers representable in the chosen precision. The test vectors in UCBTest are represented in hexadecimal form. For a complete description of UCBTest, see [Hou+88].

The third tool which IeeeCC754 is based upon is a test suite developed by J.T. Coonen [Coo84]. It shares the idea of difficult test cases with UCBTest and consists of a driver program and a large database of test vectors which are stored in a precision independent format. To execute these tests, the driver program first converts the test vectors into (binary) floating-point numbers in the target precision (single, double, or extended). The database contains test cases for the basic operations and the different conversions between floating-point formats and integer formats (see Section 1.2.1). [VCV01a] notes that "when decoding the format independent set of test vectors from [Coo84] into double precision representation, the intersection with the battery of hexadecimal double precision vectors from UCBTest is rather large."

A fourth tool should be briefly mentioned as some of its ideas were used while designing IeeeCC754: the NAG Floating-Point Validation package FPV [Cro+89], which is a commercial package developed for testing floating-point implementations. It "creates an extensive number of test operands by varying a limited number of floating-point patterns ..." [VCV01a]. These patterns are identical for the supported operations +, −, ×, /, and square root.

2.2 IeeeCC754

IeeeCC754 is an extensive test tool written in C++ to check floating-point implementations for conformity with different aspects of IEEE 754 and IEEE 854. It was developed at the University of Antwerp by a team led by B. Verdonk and A. Cuyt [VCV01a; VCV01b] with the idea to combine the approaches of the four test tools described in Section 2.1 (which are "rather complementary in nature" [VCV01a]) into a driver program that has access to a "very large set of precision independent test vectors" [VCV01a]. The driver program supports a large number of parameters to control the range and precision of the floating-point numbers, the rounding mode(s) used, whether exceptions should be checked, and to handle the two different input formats (see below).

IeeeCC754 closely follows the UCBTest philosophy to check a floating-point implementation's IEEE-conformity by executing test vectors which are difficult to round. In order to provide a comprehensive testset and to support arbitrary precision and exponent ranges, the testsets are described in a precision independent format largely based on the format Coonen used in [Coo84]. Additionally, the

format has been extended to better describe test vectors used to check conversions between different floating-point (and integer) formats. For a full description of this format (which we will refer to as *Coonen format*), see [VCV01a; VCV01b]. All UCBTest test vectors have been converted from their hexadecimal format (which we will refer to as *UCB format*) into Coonen format. In a next step, the testsets from Coonen and UCBTest have been integrated into a large test vector database.

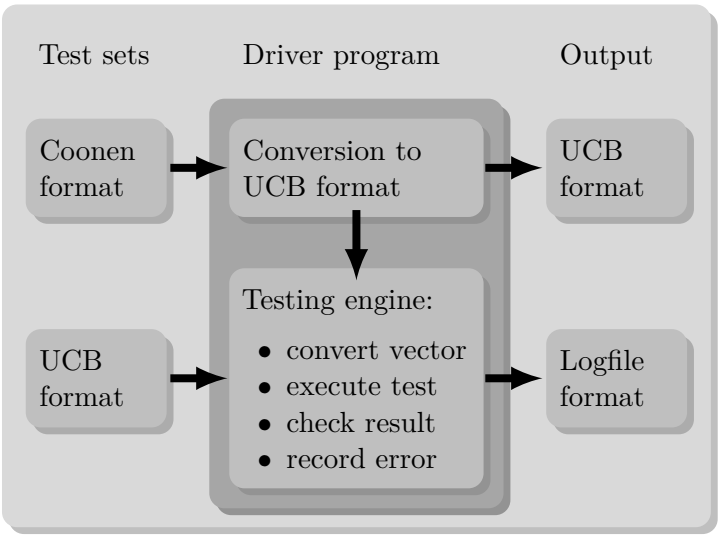

Figure 2.1: *Architecture of* `IeeeCC754`

Testing a given floating-point implementation for a particular precision works as follows (see Figure 2.1): The driver program is called with a suite of test vectors, either in Coonen (range- and precision-independent) or in UCB (hexadecimal) format and possibly some additional command line options, e. g. to select a particular rounding mode or exclude some exceptions. If the test vectors are given in Coonen format, they are internally converted into UCB format in the desired precision. At this stage, it is possible to export the converted vectors into a testset file in UCB format.

After rendering the test vectors for the target precision, every test case is considered: First, the contained floating-point numbers (operands and correct result) are encoded into floating-point numbers in the target floating-point environment's binary format, taking endianness issues into account. Then the requested operation is performed and the results are checked. This involves comparing the result returned by the environment to the known correct value, as well as checking whether exceptions were raised and comparing these to the known standard-conforming exceptions. Last, this information is used to generate logging output. In `IeeeCC754`, output is only generated for test vectors where errors were encountered. For these, the following information is logged: the operation, the operands, the correct and the returned result, expected exceptions, and finally

returned exceptions.[2]

During the execution of suitable test vectors, `IeeeCC754` not only checks for errors in the operations themselves, but also detects "which of the three IEEE-compliant underflow modes is used [...] and checks whether that underflow mode is used consistently" [VCV01a] (cf. also Section 1.2.1). In this way, the Paranoia philosophy is employed by `IeeeCC754`.

Finally, the test vector database was greatly extended with test vectors for almost all operations, partly precision dependent (especially for conversions), partly in the new precision independent format. The operations and the testsets covering them are discussed in the next section.

2.3 Testsets

In the following, we summarise how the testsets included in `IeeeCC754`'s test vector database are chosen and constructed, closely following [VCV01a; VCV01b].

2.3.1 Addition and subtraction

Addition and subtraction are basically the same operation, except for the sign of the second operand. Thus, `IeeeCC754` only includes test vectors for addition. Commutativity is implicitly checked by executing two tests for every test vector: One for $x + y$ and one for $y + x$ (if $x \neq y$).

The testset includes vectors to check for the exceptions which can be raised when executing an addition: overflow, inexact, and invalid. As described by the `IeeeCC754` authors, "the underflow exception cannot arise for the operations when the floating-point number set includes denormalised numbers, as required by the IEEE standard" [VCV01a].

The testset for addition contains test vectors with all possible combinations of the special representations (quiet and signalling NaNs, $\pm\infty$, ± 0). These are needed to check for the invalid exception which should only be raised in case of magnitude subtraction of infinities or when one of the operands is a signalling NaN.

For thorough testing of the overflow exception, many new test vectors were included. These especially contain test vectors where, depending on the rounding mode, the result is either the largest representable floating-point number (in the given exponent range) or an infinity (where the overflow exception should be raised).

[2] For example output, cf. Listing 3.3, page 82 in Section 3.3.1.

2.3.2 Multiplication

Checking multiplication is handled similarly to addition. Commutativity is again handled implicitly by reversing the order of the operands, and the overflow, invalid and inexact exceptions are covered by test vectors based on similar ideas (i. e. checking all possible combinations of special representations and using test vectors for which the result is only slightly larger than or equal to the largest representable floating-point number).

For multiplication, the underflow case has to be handled. `IeeeCC754` includes an extensive set of test vectors that check which of the three IEEE compliant detection methods (`u`-, `v`, or `w`-underflow, cf. Section 1.2.1 and [Coo84; Cuy+02]) is used and whether the applied method is used consistently. For a more detailed discussion including descriptions of the test vectors, see [VCV01a, Chapter 5.1].

2.3.3 Division

`IeeeCC754` includes test cases for all possible combinations of special representations from [Coo84] and [Hou+88], some of which raise the invalid exception, others the divide by zero exception, and others raise no exception at all.

In [VCV01a, Chapter 6.1, Lemma 1], it is shown that "floating-point operands x and y, when divided, generate an (exact) result equal to the largest representable floating-point number followed by one or more nonzero bits, such that rounding determines whether overflow will occur or not", do not exist. Thus, it is not possible to construct test cases for overflow in analogy to the addition and multiplication cases.

To the set of "several tricky test cases [...] included in the original Coonen testset" (which are based on a power series expansion, see [VCV01a]) to check inexact cases, several precision independent test vectors with denormalised operands were added based on a similar expansion.

2.3.4 Square root

When computing the square root, the invalid exception should be raised for negative numbers as well as for NaNs and infinities. Since the number of test vectors for these cases in the original Coonen testset was deemed sufficient, `IeeeCC754` only includes those test vectors for square root. It should be noted that overflow and underflow cannot occur.

For the inexact exception, it is shown that cases exactly halfway between representable floating-point numbers cannot occur for the square root ([VCV01a, Chapter 7.2, Lemma 2]). Due to the nature of iterative square root algorithms, it is tricky to get the last rounding error right by less than 1/2 ulp (see [Kah96a]). It is almost impossible to expose a flaw in the algorithm by random testing. For

this reason, IeeeCC754 includes precision independent test vectors derived from hard cases generated by the program UCBsqrttest [Hou+88].

2.3.5 Remainder

For the remainder operation, IeeeCC754 generalises test vectors from the Coonen suite and includes them in the precision independent format. The results of the remainder operation are always exact and, as indicated in IEEE 754, not affected by the current rounding mode. Operations on NaNs are the only cases where the invalid exception can occur.

2.3.6 A note on conversions

For the conversions defined in IEEE 754 and IEEE 854 (namely between floating-point formats, integers, and decimal numbers, and rounding to integral values), no test vectors in precision independent format were available before IeeeCC754. Coonen's test vectors (test vectors for single, double, and quadruple formats) have been generalised for arbitrary floating-point sets and complemented by almost the same amount of new precision independent test vectors, as well as with a large number of precision dependent vectors to test binary to decimal conversion (and vice versa). These vectors are centred around the testing of (see [VCV01b])

- the appropriate handling of special representations (signed zeroes, NaNs, etc.);
- the appropriate detection of exceptions such as overflow, underflow, and invalid (where relevant for the operation);
- exact rounding and the corresponding detection of the inexact exception.

Details about the different conversion operations are given in the following sections.

2.3.7 Conversions between floating-point formats

When a floating-point implementation supports more than one floating-point format, the IEEE standards require that conversion between all of these formats is possible. Obviously, conversion from a smaller format (i. e. a format with a smaller significand and an identical or smaller exponent range) to a larger format is always exact, whereas the result of rounding to a smaller format depends on the current rounding mode and might overflow, underflow, or be inexact.

IeeeCC754 includes test cases to check all three exceptions, as well as to check the rounding in near halfway cases and almost exactly representable floating-point results. It should be noted that not all of the test vectors encoded in precision independent format can be applied to floating-point conversions between arbitrary

formats as for some of these vectors, special conditions may apply, e. g. same exponent range and the significand in the source format at least 3 bits wider (cf. [VCV01a; VCV01b]).

2.3.8 Rounding floating-point numbers to integral values

If one assumes that the precision t and the bias B of the considered floating-point set satisfies $t - 1 \leqslant B$, performing rounding to an integral value cannot lead to overflow. As `IeeeCC754` implicitly assumes this condition (and it is met for the basic and extended formats defined in IEEE 754), only the inexact exception has to be regarded. Corresponding test vectors are included in the test suite.

2.3.9 Conversion between floating-point and integer formats

Whereas the IEEE standard defines the exact precisions of a number of floating-point formats like single and double, it only specifies that implementations "shall provide conversion operations from all supported arithmetic formats to all supported signed and unsigned integer formats" [IEEE08]. `IeeeCC754` restricts itself to the most frequently available hardware formats: integers which are 32 and 64 bits wide, with signed integers encoded in two's complement. [VCV01b] notes that vectors to test conversion between floating-point and integers formats are precision dependent more often than is the case for the other conversions. Consequently, the testsets mainly include test vectors for conversions between single and double formats and 32 and 64 bit integers. For these conversions, underflow and overflow cannot occur.[3] When converting floating-point numbers larger than the largest representable number in the target integer format, the IEEE standard mandates raising the invalid exception. Test vectors to check this case and other cases raising the invalid or inexact exception are included in the testsets.

2.3.10 Decimal to binary and binary to decimal conversion

While usually not needed inside numerical programs, conversion between binary and decimal representations of floating-point formats occur when interacting with the user, e. g. when generating human readable output or when floating-point numbers are entered textually into a program. As discussed in section Section 1.2.3, IEEE 754-2008 mandates correct rounding for these operations. Although this was not the case in the IEEE 754 and 854 standards and thus when `IeeeCC754` was developed, the authors designed test vectors for all rounding modes. Most of these are derived from algorithms designed to find numbers in the input bases 2 and 10 which, in the output case, lie close to representable numbers or exactly

[3] Underflow cannot occur for obvious reasons, and the largest representable 64 bit integer is smaller than the largest representable single number.

halfway between adjacent representable numbers [PK91]. Additionally, test cases are included where the results are exact or induce either underflow or overflow.

2.4 Results

To actually be able to execute the tests for a given floating-point environment, the driver program needs to know a) which operations are supported and b) how to translate the test vectors in UCB format to actual floating-point numbers for the current floating-point environment (especially taking endianess issues into account). The original `IeeeCC754` program described in [VCV01a; VCV01b] provides implementations for x86 processors from Intel and AMD (little endian) and Sparc processors from SUN (big endian), as well es for two software multiprecision floating-point libraries: FMLib, a `FORTRAN` library developed by D. Smith [Smi91], and MpIeee, a `C++` package which has been developed by the authors of `IeeeCC754` [Cuy+00].

It should be noted that availability especially of the conversion functions heavily depends on the user environment that is used. For conversion operations on three environments, SUN UltraSparc-II combined with `FORTRAN` and `C/C++`, and Intel processors with x87 mathematical coprocessor combined with `C/C++`, [VCV01b] gives a detailed overview which combinations of operations, precisions and rounding modes are supported.

2.4.1 Intel and AMD

At the time when `IeeeCC754` was developed and the tests in [VCV01a; VCV01b] were executed, the only available FPU in Pentium compatible (and earlier) processors was the x87 mathematical coprocessor. It is extended based, i. e. the internal floating-point format is IEEE 754's extended format (see Table 1.1). Operations performed purely in single and double format are supported by limiting the length of the significand to 23 and 52 bits, respectively. It should be noted though that the exponent cannot be limited to 8 or 11 bits and that all internal calculations are executed in extended precision.

Tests of the basic operations $+$, $-$, $*$, $/$, remainder, and square root resulted in no errors for the extended formats, whereas for single and double formats, some irregularities were encountered, such as double roundings and a change in the underflow strategy (cf. [VCV01b]). Results for the square root depended on the implementation: When using the `C` function `sqrtl` from the GNU compiler `g++` v2.95.2, superfluous exceptions were raised, which disappeared when directly calling the square root routines implemented in the x87 coprocessor via inline assembler.

Testing conversion revealed no problems for the conversions implemented in hardware, whereas conversions between binary and decimal formats (cf. Sec-

tion 2.3.10) are implemented in software and only support round to nearest. Furthermore, when converting floating-point numbers in decimal format to binary format, some numbers were flushed to zero (which should be subnormals), whereas in other numbers the last bits of the significand were not calculated correctly.

2.4.2 SUN Sparc

For the basic operations, tests were executed on SUN SuperSparc and UltraSparc workstations in single, double, and quadruple formats, with the first two being executed in hardware and the latter implemented in software [SUN97]. On all machines and for all precisions, `IeeeCC754` reported no errors for these operations [VCV01a].

As discussed in Section 1.4.6, test results for conversions are heavily dependent on the compiler choice. In [VCV01b], conversion tests were executed on an UltraSparc-II workstation with three compilers: the `FORTRAN` compiler SUN f95 and the `C++` compilers SUN CC and GNU `g++`.

SUN f95

For the `FORTRAN` 95 compiler SUN f95 from Forte Developer 6 update 1, `IeeeCC754` reported a few exception errors for conversions to 32 and 64 bit signed integers. Furthermore, the only error reported for binary to decimal conversion for all three tested precisions occurred for the negative zero -0 which was converted to a positive decimal zero.

SUN CC

When converting to 64 bit signed and unsigned integers and to 32 bit unsigned integers, some exception errors were reported by `IeeeCC754` for SUN CC. Also, another irregularity concerning zeroes was detected: When copying an integer zero to a floating-point number in round down mode, the result depended on the combination of formats involved, sometimes returning -0 and sometimes $+0$.

GNU `g++`

For GNU `g++`, `IeeeCC754` reported the same errors as for SUN CC for conversions to integers. For these conversions, some additional exception errors were detected in quadruple precision. Furthermore, quite a few errors ranging from exception errors to incorrect results were found for conversions between binary and decimal formats in round to nearest rounding mode, even though it is the only rounding mode `g++` supports for these conversions. This once again shows the influence of the compiler for those operations not implemented in hardware.

2.4.3 FMLib

In addition to the hardware platforms mentioned above, `IeeeCC754` was also applied to the `FORTRAN` multiprecision library FMLib [Smi91]. It only supports a limited set of features required in IEEE 754, e. g. only the rounding modes round to zero and round to nearest. Also, IEEE compliant special representations such as signed zeroes (± 0), denormalised numbers, and exception handling are not supported.

To check only the supported features for IEEE compliance, all test vectors involving unsupported features were excluded by calling `IeeeCC754` with the appropriate command line options. Still, `IeeeCC754` detected errors in both rounding modes and revealed that the errors were caused by using an insufficient number of guard digits in the implementation of FMLib.

2.4.4 MpIeee

Another multiprecision floating-point package has been checked with `IeeeCC754`: MpIeee, which was developed by `IeeeCC754`'s authors in the framework of the Arithmos project [Cuy+00]. The goal of developing MpIeee was to provide a high-performance multiprecision floating-point implementation in `C++` fully conforming to IEEE 754. As `IeeeCC754` was heavily used while developing MpIeee, it comes as no surprise that no errors are reported by `IeeeCC754` when applying the test vectors contained in `IeeeCC754`'s testsets to MpIeee.

Chapter 3

IeeeCC754++

IeeeCC754 was developed in order to assess the quality of a given floating-point environment. In this chapter, we present our tool IeeeCC754++ which extends IeeeCC754 in a number of crucial areas: It provides support for IEEE 754-2008 (which was not available when IeeeCC754 was written), including support for trigonometric, exponential, and logarithmic functions, extends the facilities for analysing a floating-point environment and testing results, adds a testing model to assess the IEEE-conformity of generic environments, and introduces numerous new ports for common hardware and software floating-point environments. Furthermore, we present additional facilities aimed at easing the effort needed to interpret the results of IeeeCC754++ test runs, as well as tools to enable (semi-)automated testing for a large number of floating-point environments.

As discussed in chapters 1 and 2, testing a floating-point environment can be a quite complex task. Additionally, it is not always clear what exactly should be tested: the floating-point environment as a whole, certain hardware parts or parts of an execution unit, the influence of compiler options on floating-point behaviour, the (numerical) reliability of nodes in a parallel high-performance computer, differences or even regressions between compiler versions, or CPUs with the same instruction set architecture (ISA), but different manufacturers. IeeeCC754++ aims at providing a comprehensive tool box that enables testing for all of these use cases.

In this chapter, we introduce the extensions that we added to IeeeCC754 in order to support these quite different but valid testing approaches. Furthermore, we discuss our new facilities to provide IEEE-conformity, enable deeper analysis of floating-point environments, and to offer easy extensibility for new floating-point environments. Afterwards, we briefly describe the input and output formats used by IeeeCC754++ and introduce the testing modes which we implemented in IeeeCC754++ to support the use cases described before. We conclude this chapter by introducing a framework that enables mass testing of a large number of different (floating-point) architectural features. We also describe the new analysis facilities

that help evaluating test results and introduce a variant of the testing framework which significantly eases the analysis of the influence of compiler options on floating-point performance and conformity.

For details about the supported architectures and FPUs as well as how testing these has been implemented in IeeeCC754++, we refer to Chapter 5; for a detailed analysis of their floating-point performance and conformity, see Chapter 6 which describes results for a selection of the supported architectures.

3.1 IeeeCC754++: Introducing extensions

The complexity of a floating-point environment as discussed in Section 1.4 raises the question what can and should be tested when checking the conformity of a given floating-point environment. In this section, we discuss our approaches to address different testing needs, give an overview of the features that we added in IeeeCC754++ to support these approaches, and explain the motivation that led us to the addition of the new features. We start with some general considerations on testing the IEEE 754-2008 conformity of a given floating-point environment.

3.1.1 Testing IEEE-conformity

IeeeCC754 and IeeeCC754++ check the conformity of a floating-point environment by executing floating-point operations for carefully selected operands whose results (correct floating-point number and floating-point exceptions that need to be raised) are encoded in so-called test vectors, together with the operands and the operation (cf. Section 2.2). Testing is done on testsets comprising of collections of these test vectors. The testsets can be loosely categorised into two groups: tests for numerical correctness and testing of special cases. The first group consists of test vectors whose result is difficult to compute or difficult to round (or both), mainly in the sense that the infinitely precise result lies either halfway between or really close to two neighbouring floating-point numbers. The second group covers a range of special cases that an IEEE-conforming floating-point environment must support correctly, such as overflow (i. e. handling infinities), exceptions that should be raised (e. g. in the case of overflow and underflow), and handling NaNs. The quality of the IEEE-conformity assessment depends crucially on the choice of test vectors – only a comprehensive set of test vectors leads to a meaningful conformity assessment. This problem arises for every application for which it is only feasible to execute a limited number of test scenarios in order to assess correct functionality of the whole application. As a result, with poorly chosen combinations of IeeeCC754++ test vectors, only those parts of IEEE-conformity that these test vectors cover can be evaluated. For a more detailed description of the testsets employed by IeeeCC754++ and the reasons why they form a solid basis for IEEE-conformity assessment, we refer to Section 2.3 and Chapter 4.

Another area to consider is the evaluation of the results of a test run with a carefully selected testset, assuming that it covers a sufficiently broad range of test vectors: If all tests can be successfully executed (in the sense that executing the operation encoded in that test vector on the given operands returns the correct result and the correct exception flags), we assume that all parts of the floating-point environment are as carefully (and correctly) implemented as those parts that were actually tested.

The situation is different when errors are encountered. In that case, the errors must be carefully analysed: Since test vectors are chosen with a specific target area in mind (such as testing special cases, see above), one must carefully examine the operations which could not be successfully completed and the type of error that occurred during execution. If e. g. all (or a large percentage of) test vectors that employ roundTowardPositive return an error, but all other test vectors execute successfully, one would conclude that the implementation of the roundTowardPositive rounding mode is flawed. Since examination of a large number of environments for errors is a tedious process, `IeeeCC754++` offers approaches to ease this process:

- Specialised testing modes and output formats: `IeeeCC754++` contains a number of testing modes and corresponding output formats that cover testing of different aspects of a floating-point environment, such as a verbose mode that produces parsable, but still readable error output that is very detailed, or checksum and fingerprint modes which return a single output value for a full test run. For details, see Section 3.3.

- The evaluation framework: an extensive framework that can summarise the errors returned by an `IeeeCC754++` run in verbose mode. By using the evaluation framework, it becomes easier to recognise specific erratic behaviour of the test floating-point environment (such as rounding modes that are not supported or no exceptions available). The evaluation framework is explained in Section 3.4.

3.1.2 Testing the user environment: Default mode

Goal Provide a tool that can be used in an easy way to assess the IEEE-conformity of a given user environment without deeper (prior) knowledge of the underlying floating-point implementation.

Reasoning Before adoption of IEEE 754 as a common standard regulating floating-point calculations, knowledge of the underlying platform and the pitfalls of its floating-point implementation was vital in designing robust and stable numerical algorithms. Although IEEE 754 improved this situation in the sense that common (and well-known) floating-point formats and high-quality implementations of floating-point operators are used on all major platforms, actual knowledge of the

platform and numerical pitfalls have become less common as researchers simply assume that their target floating-point environment behaves "as expected", i. e. IEEE-conformity is relied upon without checking this assumption.

In an ideal academical world, developing numerical algorithms should depend neither on the development nor on the target user environment, and the choice of programming language or the compiler used should not influence the performance of the algorithm with regard to numerical quality. However, in practice all of these choices can lead to subtle numerical problems which might go undetected since the differences between the different resulting environments (in the sense of Definition 1.5) are difficult to recognise or test (and even worse, these environments might not be checked for comparability at all).

One goal of `IeeeCC754++` is to provide a testing tool that checks the IEEE-conformity of the *default environment* a researcher uses. The typical "black box" experience of such an environment is as follows: The user writes an algorithm in the programming language of choice, uses one of the compilers available in that user environment to generate an executable, and runs the resulting binary file to retrieve results. Usually, the code is executed on the local environment, at least for development and debugging purposes. The subtleties of how exactly floating-point operations are executed, e. g. in different FPUs or in software, are mostly lost to the typical researcher. Actually, one could even argue that this is the ideal approach to numerical computing, assuming that all parts of the execution chain are IEEE-conforming.

Realisation With these considerations in mind, we can define the term default user environment as used in this thesis:

Definition 3.1. A *default user environment* is a floating-point environment consisting of the following parts:

- a programming language used to write a numerical program,
- a computing platform used to execute this numerical program,
- and a compiler (possibly including compiler switches) that translates the user program into code which can be executed on that platform. ◊

Note that this definition is deliberately vague concerning the underlying computing platform. It focuses on the part of the development process that a typical researcher experiences: the programming language and the compiler, since these are usually the areas that can be chosen by the researcher and which she uses to interface with the computing platform.

In order to test such a default user environment for IEEE-conformity, we implemented the *default mode* which employs a specific architecture port (cf. Sections 3.3 and 5.1) that tries to mimic the default user experience as much as possible, i. e. it employs only the default arithmetic operators and mathematical

functions as found in the programming language. The choice of the exact execution point of floating-point operations is left to the compiler. In that way, it provides the user with a tool to test her default environment with the exact settings she or he uses in the normal development process.

When running on a given user environment, the results of testing depend on the underlying computing platform itself and on the choice of compiler and compiler options. Since the environment is seen as a "black box", the default mode focuses mainly on the influence of the compiler and the compiler options employed. This view is crucial in understanding testing results as the compiler might choose different floating-point execution units depending on the chosen options.

Sometimes, the default user environment consists of more than one platform: For development purposes, a local platform like the researcher's workstation might be used, whereas the resulting algorithms would be run on a remote platform, e. g. on a supercomputer such as JUQUEEN [JSC17a]. In this case, the default mode can be employed in two variants: The local platform is regarded as the default user environment and tested with the default mode. Afterwards, the locally generated `IeeeCC754++` executable can be transferred to the remote platform and used for testing the default environment on that floating-point environment. Comparing the testing results from both environments yields interesting insight into the degrees of IEEE-conformity of the environments and, more importantly, into the differences in conformity between the environments. More details regarding this approach can be found in Section 3.1.4.

The second approach views the local and the remote platform as two default user environments. This means that both platforms should be individually tested for conformity, including compilation of `IeeeCC754++` on both platforms, possibly with different compilers or compiler switches. Note that this approach allows for the local and remote platform to be of a different computer architecture, e. g. when developing code on a typical x86 computer which is ultimately intended to be run on a POWER based supercomputer such as JUQUEEN.

It is important to understand the default mode approach as a holistic approach to conformity testing: Its scope is limited to checking the floating-point quality of a researcher's floating-point environment as the researcher experiences it. If checking single components of the user environment is desired, specialised architecture ports such as those introduced in Section 3.1.3 below and Chapter 5 should be used.

Since `IeeeCC754++` is written in `C++` and `C` and `C++` represent the most widespread programming languages in the scientific computing world, the default mode only supports these two languages. However, for all programming languages which support calling functions from `C/C++`, it is possible to implement custom architecture ports that check the operators supported by that language. `IeeeCC754++` itself comes with support for testing the floating-point facilities of

Java (see Section 5.7.1).

For a detailed description of how the default mode has been implemented in IeeeCC754++, see Sections 3.3.3 and 5.1.

3.1.3 Testing parts of a floating-point environment

Goal Provide means to not only test the default user environment, but also certain parts that, put together, constitute the floating-point environment a user experiences.

Reasoning The scope of IeeeCC754++'s default mode that was presented in the previous section is limited to checking the IEEE-conformity of a given user environment "as is", i. e. as a user would experience it. Often, it is more desirable to check specific parts of a floating-point environment, e. g. when deciding which of the available execution units of a given floating-point environment should be used or when developing new FPUs or software floating-point libraries. IeeeCC754++ strives to provide a testing tool that covers as many aspects of a floating-point environment as possible (see also Section 1.4):

Realisation In order to provide for such a diverse testing tool, IeeeCC754++ offers the following features to check parts of a floating-point environment:

- *Single FPUs*: The most obvious execution targets are the dedicated floating-point execution units found in most modern CPUs. In order to check the IEEE-conformity of an FPU regardless of the compiler used, the routines for testing most FPUs are implemented via intrinsics or inline assembler commands (see also Chapter 5).

- When floating-point operations are executed via *software libraries*, the operators in these libraries are accessed via function calls. Additionally, if a library uses custom floating-point data formats, conversions between IeeeCC754++'s internal floating-point format and the format native to the library are implemented.

- *Accelerators*: Since accelerators such as GPUs play an important role especially in supercomputing, IeeeCC754++ supports checking accelerators via special architecture ports (cf. e. g. Sections 5.2.2 and 5.5,

- *Hardware architectures*: IeeeCC754++ supports a number of different hardware architectures, such as x86, POWER, or ARM. Although the checking process will be done on actual hardware platforms, it is important to note that these hardware platforms (e. g. a certain type of processor) constitute specific implementations of an ISA. IeeeCC754++ implements support for ISAs and can therefore be run on all platforms that implement that ISA, such as Intel or AMD processors implementing the x86 ISA.

- IeeeCC754++ also includes support for some *programming languages* and *interpreters*. As discussed in the previous section, these languages include C, C++, and Java.

- As discussed in Section 1.4.8, some parts of a floating-point algorithm might be offloaded to the network fabric (in case of the user environment being a parallel computing environment such as most HPC supercomputers). In that case, systematic and deterministic testing of these operations is out of scope of IeeeCC754++'s usage model. However, IeeeCC754++ includes an architecture port for MPI that can be used to check the IEEE-conformity of MPI collective operations. For details, see Section 5.6.

3.1.4 Testing distributed floating-point environments

Goal Introduce concepts for testing the IEEE-conformity of computing environments consisting of more than one platform.

Reasoning Especially in scientific computing, the typical floating-point environments used to execute large numeric simulations consist of more than one platform, which we call *distributed environments* or *distributed computing*. This can e. g. be the case when executing an application on several computers in a local network, on an HPC supercomputer, or in a cloud environment. Note that we use the term distributed computing differently and in a much looser manner as compared to the traditional sense where distributed computing is a field of computer science that studies "a model in which components located on networked computers communicate and coordinate their actions by passing messages" [WIK17i].

Analysing the floating-point behaviour and the IEEE-conformity when more than one floating-point user environment is involved poses unique challenges: Sometimes, applications are developed in one user environment, but are targeted to be executed on a large supercomputer that features a completely different user environment. In that case, both environments have to be evaluated separately to gain enough trust that executing numerical code on both environments yields comparable results (see also Section 3.1.2). If on the other hand several identical (or at least similar) user environments are involved, one can employ a more advanced checking model that generates a local IEEE-conformity result by executing IeeeCC754++ in the local environment before transferring the IeeeCC754++ executable to the remote environment, repeating the testing process in that environment, and comparing the local and remote results.

Realisation For the approach of comparing results for the local and remote environment, we assume the researcher is familiar with the floating-point environment in which she develops the target numerical application. The "numerical compatibility" can then be assessed as follows: First, a suitable architecture port,

testing mode, and testset have to be chosen (e. g. the default mode or a hardware architecture port for the involved environments such as x86, together with e. g. a testset that covers basic operations in double precision, cf. Section 4.1.2). Additionally, the compiler used to build the target application as well as the compiler switches used during compilation have to be known. `IeeeCC754++` can then be built with these settings and executed in the local environment, yielding a result file. As a second step, the newly built `IeeeCC754++` executable is transferred to the remote environment and executed with identical settings. Now, evaluating the numerical comparability is equivalent to comparing the results generated on both platforms.

This approach is also suited to verify that a given number of nodes behave identically with regard to the floating-point environment, e. g. on HPC supercomputers consisting of a large number of identical computing nodes: One node is used to generate an `IeeeCC754++` executable. This executable is then run on all nodes that are to be evaluated, and the results are compared. If indeed all nodes yielded identical testing results, it is reasonably safe to assume that the nodes also behave identically when executing numerical applications. Furthermore, the concept of generating a IEEE-conformity result in a known environment and comparing with results generated with an identical executable in different environments can be used as a safety measure before starting the target application: If the results are not identical, one must be aware that results generated in the remote environment might not be of the same quality as those produced in the local environment. If desired, the application could even be aborted prior to execution. In fact, this approach has been used with an early version of `IeeeCC754++` in the LHC Computing Grid ([WLCG], cf. [Mül+07]).

Of course, this approach can be applied to a wide range of distributed environments: HPC computers, grid and cloud computing, or distributed computing in the original sense (cf. [WIK17i]).

`IeeeCC754++` provides two testing modes that are specifically target at the presented approach: the *checksum mode* that, instead of using large logfiles, generates a condensed binary report usable for faster comparison of environments, and the *fingerprint* mode that yields a fingerprint of only a few bytes suitable for e. g. being stored in a database. For details on these modes, see Section 3.3.4.

3.1.5 Supporting arbitrary floating-point environments

Goal Supply means to test arbitrary environments and provide a framework that enables easy extensibility for platforms not yet supported.

Reasoning `IeeeCC754` already implemented support for a number of floating-point environments such as x86 and SUN Sparc, split into basic operations and conversions (cf. Chapter 2). To cater for different execution options for the remainder and square root functions on x86 platforms (`C++` function call vs.

assembler implementation), one set of the respective functions was placed in the implementation for basic operations while the other was placed inside the conversions implementation.

However, this selection does not reflect the currently available computing platforms (such as POWER or ARM processors or NVidia or ARM GPUs) nor relevant floating-point software libraries (such as MPFR). Arguably, supporting a comprehensive set of current floating-point environments is a moving target at best and simply impossible at worst, due to the rapidly changing computing landscape (from a hardware perspective as well as from a software or operating system perspective). Additionally, placing different function calls for the same operator into parts that were originally logically divided into arithmetic operations and conversion feels counter-intuitive.

Therefore, we redesigned the internal structure of `IeeeCC754++` in a way that it supports extensions for arbitrary new computing platforms. A wide range of current hardware and software floating-point environments is already supported by `IeeeCC754++`, and the source code is structured in a way to enable future additions of newly designed platforms or FPUs.

In fact, this feature is already demonstrated in the current implementation: As of writing this thesis, no hardware is available for the new ARM SVE SIMD instruction set. However, with the help of instruction simulators, `IeeeCC754++` includes an implementation able to check the IEEE-conformity of ARM SVE hardware once it is released. For details, see Section 5.3.2. The same approach was used in the past to add an AARCH64 implementation to `IeeeCC754++` prior to hardware being available.

In Section 3.1.3, we shortly discussed a variety of floating-point environments which differ in the details how operations in these environments are implemented and executed, with the differences sometimes being substantial. Supporting e. g. accelerators generally requires using specific toolkits or libraries, employing function calls from these libraries, and even might require initialising the device driver in order to execute floating-point operations on an accelerator. On the other hand, checking the default user environment only requires the availability of a suitable `C++` compiler. With this in mind, the old `IeeeCC754` implementation model is clearly not sufficient to support such a variety of different floating-point environments.

Realisation From a conceptional point of view, all execution points of some floating-point operation can be considered as part of some FPU, be it a software routine, a function call to an accelerator library, or an instruction inside a hardware FPU. Consequently, all operations with similar execution points can be collected into one *logical* FPU. This approach renders checking the IEEE-conformity of such an FPU feasible by collecting similar operations into one logical unit which can be handled in a unified manner, i. e. the implementation, initialisation, and execution of the operations inside a logical FPU are done in a similar manner.

Taking this concept further, several FPUs with similar features can be combined into one architecture port. `IeeeCC754++` makes extensive use of this approach to support all varieties of floating-point environments, according to the following definitions:

Definition 3.2. An *FPU* (also called *logical FPU*) inside `IeeeCC754++` consists of a group of floating-point operations that are called in an identical manner and in identical execution units and which can be compiled into one executable. Furthermore, it must be possible to handle the setting and retrieval of rounding modes and exception flags in an identical manner. ◊

The meaning of FPU in this thesis depends on the context: It might refer either to a specific hardware execution unit inside a processor or to a collection of floating-point operators in the sense of this definition. We therefore annotate FPUs with either *hardware* or *logical* if the meaning of FPU might be ambiguous.

Definition 3.3. An *architecture port* inside `IeeeCC754++` consists of one or more `IeeeCC754++` (logical) FPUs. If the context is clear, it is simply called *architecture*.

The obligatory FPU contained in every architecture port is called the *main FPU*. Additional FPUs must be compilable into one common executable together with the main FPU. ◊

Note that Definition 3.3 does not state that all additional FPUs inside an architecture must be compilable into one executable at the same time, i. e. the case that only subsets of FPUs are compilable together with the main FPU is explicitly allowed. In other words, a newly added FPU must only be compilable together with the main FPU. Also note that an architecture in `IeeeCC754++` sense can be almost anything that comprises a floating-point environment, e. g. a "real" hardware platform or ISA such as x86, POWER or ARM, a software floating-point library, or a programming language/interpreter combination such as `Java`. For more details, we refer to Chapter 5.

With these concepts, building `IeeeCC754++` for one user environment and checking that environment for IEEE-conformity works then as follows:

- The first step is to choose an architecture port appropriate for the environment to be tested. Furthermore, the subset of logical FPUs for which code should be compiled must be selected.
- A suitable compiler must be chosen.
- Then, the build system (see Appendix A) is set up for this choice of architecture, FPUs, and compiler, possibly incorporating necessary compiler switches or libraries which must be linked into the resulting `IeeeCC754++` executable. Afterwards, the `IeeeCC754++` source code is compiled.

- Now IeeeCC754++ is prepared to execute IEEE-conformity checks for the selected FPUs. IeeeCC754++ is executed for every combination of FPU and testset (cf. Section 2.3 and Chapter 4), and the resulting output files are available for further analysis.

Note that although one IeeeCC754++ binary is built for each architecture port, the actual testing is always performed on (logical) FPUs. As an example, the default mode discussed in Section 3.1.2 is implemented as an architecture that can be built on any user environment with a suitable C++ compiler. C99 and C++11 support is built into this architecture as additional FPUs. All three FPUs (main, C99, and C++11 FPUs) are available for testing. For details on the implementation, see Section 5.1.1.

Depending on the floating-point environment, it is possible to build several IeeeCC754++ executables with different architectures for that environment. Taking once more a typical x86 workstation as an example, an executable for the default architecture can be built to test the default user environment as well as a specific x86 architecture executable capable of testing e. g. the SSE FPU inside that workstation

A comprehensive list of the architectures and their FPUs which we implemented in IeeeCC754++ can be found in Chapter 5, together with detailed descriptions of their implementation. Since it is impossible to supply implementations for all floating-point environments, IeeeCC754++ has been restructured in a way that support for new environments can be implemented with moderate effort. In order to be able to compile IeeeCC754++ for arbitrary environments, a new build system has been added to IeeeCC754++, including support for the floating-point environments currently implemented in IeeeCC754++. The process of implementing new architectures and FPUs is explained in Appendix B. Details on the build system and how it can be extended to build code for the new architecture can be found in the next section and Appendix A.

3.1.6 Building for arbitrary environments

Goal Support building IeeeCC754++ on all supported environments and enable compilation on future environments.

Reasoning As described in Section 2.4, IeeeCC754 supports only a limited number of floating-point environments. Therefore, it was sufficient to use a custom handwritten configure script to build the driver program for these environments. For the development of IeeeCC754++, this represented a major restriction especially in terms of portability since for almost every combination of CPU or software library, FPU, compiler, and host (i. e. the computer where tests were executed), it would have been necessary to add custom configure/build code.

Providing a testing tool that supports a virtually unlimited number of floating-point environments poses a number of challenges:

- Every supported environment is implemented as an architecture including at least one FPU. Especially accelerators such as GPUs or software floating-point libraries need specific header files and libraries in order to be compiled and executed. This means that build system setup for every architecture needs code specific to the chosen environment and FPUs.

- Furthermore, the software environment can vastly differ between platforms, even when the same architecture port is used. The build system must cater for these specific needs or at least provide means to configure the build system in a way that `IeeeCC754++` can be built.

- Finally, different compilers need to be accounted for: Header files might have different names, or different compiler switches might be necessary to build `IeeeCC754++`.

Realisation In order to overcome these challenges, a build system that supports a huge number of user environments and provides flexible setup possibilities must be used. For maximum portability, the Autotools family of configure/build scripts [WIK17o; GNU16a] was chosen. It is supported on virtually any UNIX-like system and provides the flexibility to add setup options as needed. For a detailed description of the `IeeeCC754++` build system, see Appendix A.

3.1.7 Large-scale testing and analysis – the evaluation framework

Goal Enable testing of a large number of combinations of floating-point environments, architectures and FPUs, compilers, and testsets with reasonable effort.

Reasoning For most architectures that are implemented in `IeeeCC754++`, there exist several different FPUs that test either different aspects of a hardware FPU or the different hardware FPUs available in that architecture. Furthermore, in most floating-point environments, `IeeeCC754++` can be built for several different architectures, such as the default architecture, an architecture reflecting the underlying platform's ISA, an architecture for an accelerator such as a GPU, or architectures for one or more software floating-point libraries. Moreover, different compilers (or compiler versions) can be used to compile `IeeeCC754++` for these environments. Finally, different testsets can be used to check certain aspects of floating-point environments.

Checking all these combinations of architectures, FPUs, and compilers for IEEE-conformity can thus become quite a tedious and time-consuming task, let alone the analysis of the resulting logfiles. In order to provide a tool that covers creation and execution of a large number of test runs as well as facilities to assist

with deep analysis of the results of this large-scale execution, we developed the evaluation framework.

Realisation In order to help with the evaluation process of a large number of result files, the evaluation framework features extensive parsing and analysis capabilities. For maximum flexibility, the framework offers the possibility to build (and evaluate) arbitrary combinations of architectures, FPUs, compilers, compiler options, and testsets. The results of the test runs can be evaluated and analysed with one (or more) of the supplied evaluation functions which aggregate and summarise the error information contained in the log files. The included evaluation functions cover a wide range of evaluation needs, such as success rates for the individual operations or rounding modes, error types that were encountered, summaries for groups of operations, or a list of all errors that occurred. Furthermore, the analysis facilities can easily be extended with custom analysis and aggregation functions tailored specifically to the user's evaluation requirements.

The evaluation framework is described in detail in Section 3.4, including the input files that drive the different tasks performed by the evaluation framework. All results presented in Chapter 6 have been generated and analysed using this evaluation framework.

In addition to the evaluation framework, several scripts have been developed that automate the (mass-)generation of test description files and ease the execution of a large number of evaluation framework runs. Finally, a log file viewer GUI (Graphical User Interface) is provided that enables convenient access to the log and analysis files generated by `IeeeCC754++` and the evaluation framework. For details, see Section 3.4.3.

3.1.8 Studying the influence of compiler options – the optimisation framework

Goal Provide a tool to conveniently study the influence of compiler options on application performance and floating-point behaviour.

Reasoning When "fixing" the underlying platform with the default mode's black-box approach (cf. Section 3.1.2), the reasons for differences in floating-point performance lie in the choice of the compiler and its options. When limiting oneself to one compiler, the default mode can therefore be used to study the influence of compiler options by running `IeeeCC754++` with different sets of options. Since the influence of compiler options on floating-point performance and IEEE-conformity is an interesting subject by itself and manually analysing even a moderate number of log files generated with `IeeeCC754++` in default mode is tedious, we developed a specialised tool called the optimisation framework which offers an automated

way of exploring the effect of compiler options not only using the default mode, but every architecture port that is supported by `IeeeCC754++`.

Realisation The evaluation framework is mainly focused on varying the parameters architecture, FPU, and compiler (and compiler version) as well as easing the subsequent analysis. However, an especially interesting topic is the influence of compiler options, mainly with two regards: how can performance of a user program be increased (or, in other words, how can the execution time be reduced), and the effect of these options on floating-point conformity, i. e. how compiler options that increase performance affect the accuracy and IEEE-conformity of a floating-point environment. Using the evaluation framework to study this influence for a given architecture and compiler suffers from two drawbacks:

- Essentially, for every set of compiler options, a new set of input files needs to be generated, resulting in a huge number of files and test runs which have to be handled.
- The effect of the chosen compiler options on application performance (especially the effect on the application's runtime) have to be studied externally, i. e. the application has to be compiled with the same compiler options as used in the corresponding `IeeeCC754++` run, and the runtime has to be recorded manually.

In order to overcome these drawbacks, we developed the optimisation framework which constitutes a variant of the evaluation framework specifically tailored to study the influence of compiler options. Besides offering simple facilities to test a large number of compiler option combinations, it offers the possibility to automatically compile a user application with the chosen combination of compiler options, execute it, and record the runtime. Afterwards, it is possible to choose the most appropriate combination using either one of the pre-supplied fitness functions or a custom fitness function written by the user. Since potentially a huge number of individual test runs are necessary to study the effect of even a reasonably sized set of compiler options, the optimisation framework avoids the creation of task description files to drive execution of the compile, test, and evaluation tasks. Rather, it features an efficient implementation making use of in-memory data structures.

We have given an in-depth introduction of the optimisation framework in Section 3.5; for selected results, see Section 6.9.

3.1.9 Testing modes

Goal Provide a flexible tool which enables testing the different aspects that constitute floating-point environments, such as the default user environment, different parts of the floating-point stack, or distributed environments (cf. Section 1.4).

Reasoning To support the large amount of use cases and provide for a flexible testing tool, it was necessary to carefully structure `IeeeCC754++` in a way that the code base is easily maintainable and extensible. This has been achieved by introducing testing modes together with specialised input and output formats, targeted at specific use cases. Figure 3.1 gives an overview of the testing modes and the different input and output formats that provide coverage of the mentioned wide range of testing applications. A short specification of the testing modes is given below, while a detailed description of the modes and input and output formats can be found in Section 3.3.

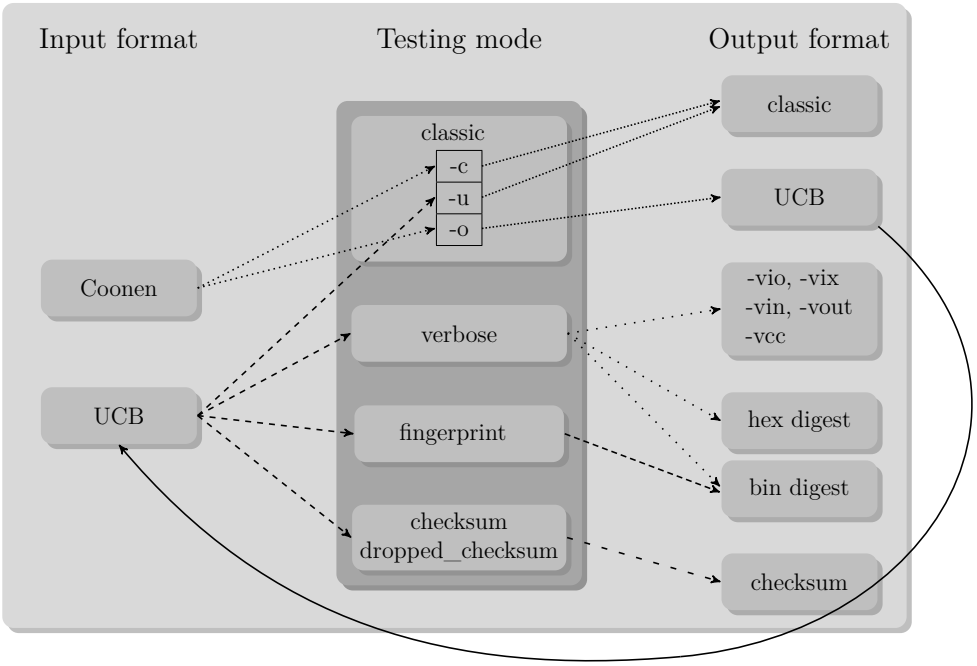

Figure 3.1: *Overview of `IeeeCC754++` input and output formats and testing modes. Note that the UCB output format of the classic conversion mode is identical to the UCB input format that is used as input for most of the testing modes.*

Realisation In order to support the afore mentioned quite different use cases, we developed and implemented *testing modes* targeted at specific use cases:

- In *default mode*, the current user environment is tested with a "black box" approach, possibly influenced by compiler switches (cf. Definition 1.5, page 25 and Definition 3.1, page 54). This can be used to check the platform's IEEE-conformity under the current user's build settings.

  Additionally, the default mode includes facilities to check for the compatibility of several versions of programming language standards, in particular `C99` and `C11`. These are implemented as logical FPUs in the default architecture, see Section 5.1.1.

- For backward compatibility with `IeeeCC754`, the three different original testing modes (featuring the original `IeeeCC754` calling syntax) are conserved: testing with both supported test vector formats and converting from precision independent to precision dependent format. This mode is called *classic mode*.

- The *verbose mode* is designed as the most general testing mode. Its strength lies in a carefully designed output format that contains detailed information on the executed test vectors, including some analysis of the type of error, which is easily parsable and human-readable at the same time.

- The *checksum* and *fingerprint* modes generate short summarised information of the IEEE-conformity of a specific floating-point environment. They are particularly suited for use in distributed computing environments (see Section 3.1.4).

Note that the testing modes are independent of the floating-point environment that `IeeeCC754++` is built for (with the exception of the default mode which is a combination of the verbose mode with the default architecture port, cf. Section 3.3.3). This means that all architecture ports described in Chapter 5 can be used together with the verbose, classic, checksum, and fingerprint modes. A detailed description of the testing modes is given in Section 3.3.

3.1.10 Input and output

Goal Choose and design input and output file formats that support all testing needs covered by `IeeeCC754++`.

Reasoning `IeeeCC754++` follows `IeeeCC754`'s philosophy and employs an extended driver program to test the floating-point environment. The extent to which the IEEE-conformity of a given user environment can be assessed then depends on the test vectors fed into the driver program (cf. Section 3.1.1), and consequently, the input file format must enable efficient input of suitable test vectors. Furthermore, the information gathered during test execution must be provided in a way that enables efficient analysis, ideally for manual and automated inspection at the same time. In other words, output should be easily human- and machine-readable. The various testing needs introduced in the last sections such as evaluation of a default user environment (Section 3.1.2) or analysis of the IEEE-conformity of distributed environments (Section 3.1.4) are supported by introducing output formats tailored to the particular testing focus. Differentiation for the different use cases is then mainly achieved via two complementary means: on the one hand using specialised testing modes together with corresponding output formats and on the other hand calling architecture ports which represent `IeeeCC754++`'s implementations for a given hardware or software FPU (cf. Chapter 5).

Realisation In contrast to the presentation of the other additions to IeeeCC754++, we deviate here in the description of the realisation of IeeeCC754++'s input and output facilities by first introducing the corresponding file formats and then discussing in detail the reasoning behind choosing and developing these specific formats and their realisation in IeeeCC754++. These facilities consist of two input file formats containing the test vectors which perform the actual IEEE-conformity testing and various output formats which support the different testing use cases.

Input: test vector formats

IeeeCC754++ knows two formats for test vectors that are fed into the driver program via input files: the Coonen format which features range- and precision-independent descriptions of the test vectors, and the UCB format in which each test vector is rendered in hexadecimal notation for exactly one combination of one precision and one rounding mode.

In order to be applicable to the largest possible range of precisions and rounding modes, the native format used to encode IeeeCC754++'s vectors is the Coonen format. However, as described in Section 2.2, all test vectors are internally converted into UCB format before executing the corresponding floating-point operation (see also Figure 3.1). Therefore, the UCB format was chosen as input format for all extended modes that we added to IeeeCC754++ due to the following reasoning:

- Using the UCB format improves efficiency as input in Coonen format is always internally converted to UCB format before testing. The overhead of creating the same UCB test vectors every time IeeeCC754++ is executed can be avoided. This is especially relevant when a larger number of architectures, compilers, or FPUs are tested.

- When checking for IEEE-conformity, only a moderate number of floating-point formats needs to be considered: Many floating-point architectures only support single and double precision formats, and some additionally one of extended or quadruple precision (cf. Table 1.1). In the last years, use of the half precision format has become popular especially in machine learning applications. When checking for IEEE-conformity, it is therefore sufficient to regard at most five different floating-point formats, and it is feasible to generate the needed test files from Coonen format in advance and use them as input files. If desired, all input files can even be integrated into one single input file.

- Finally, when reading input from UCB encoded files, the line number of the current test vector can be included in the testing output. This renders mapping between the result and corresponding input vector trivial. With Coonen format, this mapping would be ambiguous since one input line in

Coonen format can result in up to ten different test vectors (taking the five rounding modes and commutativity into account).

Note that the syntax of Coonen and UCB files as employed by `IeeeCC754++` has been extended compared to the formats used in `IeeeCC754` in order to reflect the new features added with `IeeeCC754++`, cf. e. g. Section 3.1.11. For a detailed description of the expanded syntax, see Section 3.2.1.

Output: specialised output formats

During the testing process, the original `IeeeCC754` generates output on individual test vectors only when discrepancies between the computed and the correct result (including exception flags) are detected. This is done in an informative, but tedious plain text format not particularly suited for further (automatic) analysis since it cannot be parsed efficiently. To support all features described in this section, we designed and implemented a number of different output formats targeted at the different use cases:

- The *plain* output format is basically the plain text format used in original `IeeeCC754` logfiles. The output is enhanced compared to `IeeeCC754` to account for `IeeeCC754++`'s extended analysis facilities (see Section 3.1.12).

- The *verbose* output format has been designed as an all-purpose format that enables convenient parsing, but is still easily human-readable. It is mainly used in the verbose and default testing modes (Sections 3.3.2 and 3.3.3) and provides the output format used in the evaluation and optimisation frameworks (cf. Sections 3.4 and 3.5).

- The *checksum* and *fingerprint* output formats are short formats that can be efficiently used together with the distributed computing modes (see Section 3.1.4).

Since these specialised output formats are closely related to the corresponding testing modes, we postpone the discussion of the formats including their syntax and examples to Section 3.3.

3.1.11 Support for IEEE 754-2008

Goal Update the feature set for testing IEEE 754 conformity and analysing the underlying floating-point environment to the latest revision of the standard, namely IEEE 754-2008.

Reasoning As noted in Section 1.2.3, the IEEE 754-2008 standard was formulated in a way that all floating-point implementations conforming to IEEE 754 also conform to IEEE 754-2008. As a consequence, the testing facilities implemented in `IeeeCC754` with IEEE 754 in mind still represent a valid approach to test IEEE 754-2008, i. e. the correct results stored in `IeeeCC754++`'s test vectors are still correct results in the sense of IEEE 754-2008. Furthermore, the internal analysis facilities such as detection of the underflow mode used in the tested user environment or the detection of returned exceptions still work as expected, even under the new standard.

However, IEEE 754-2008 adds some significant changes to the older standards, such as new operators and a new rounding mode.

Realisation `IeeeCC754++` reflects the additions to IEEE 754-2008 by implementing the following features:

- *Test vectors with three operands*: All operations that are required in IEEE 754 have at most two operands. `IeeeCC754` supports these operations by having two operands in each test vector. When the operation only needs one operand (like e. g. `sqrt`), a zero is recorded as second operand, and the value is simply ignored when parsing the corresponding test vectors.

  In order to maintain backward compatibility and leave current test vectors untouched, `IeeeCC754++` extends the parser in a way that the test vector description can contain two or three operands – two for all operations with one or two operands (as before) and three for all operations that need three operands (currently only `fma`, see below).

- *RoundTiesToAway*: IEEE 754-2008 adds a fifth rounding mode called roundTiesToAway (see Section 1.2.3). While this rounding mode is only mandatory for decimal floating-point operations, it is a valid rounding mode also for binary floating-point implementations. The parser and the test vector syntax of `IeeeCC754++` have been enhanced to allow for roundTiesToAway. Furthermore, when a test vector in Coonen format is applicable to all rounding modes, an additional test vector in UCB format is generated with roundTiesToAway as rounding mode.

  In addition to enhancing the parser, the existing test vector files have been extended by test vectors that check rounding in roundTiesToAway mode.

- *Fused multiply-add* (`fma`): `IeeeCC754` supports the operations that are required in IEEE 754, which are the basic operations $+$, $-$, $*$, $/$, remainder, and square root, and conversion operations from floating-point formats to floating-point, integer, and decimal formats. `IeeeCC754++` adds support for `fma` which is required in IEEE 754-2008.

- *New operators*: In addition to `fma`, IEEE 754-2008 recommends a number of arithmetic functions to be correctly implemented, i. e. it recommends that IEEE-conforming floating-point environments implement these operators with correct rounding. The recommended operators include trigonometric, exponential and logarithmic as well as power functions, cf. [IEEE08, Section 9.2]. All of these functions are now supported in `IeeeCC754++`. For a list of the newly supported operands and the newly added test vectors, we refer to Table B.1, page 325, and Chapter 4.

- *Half precision*: IEEE 754-2008 adds support for a 16 bit floating-point format called *binary16* or *half* since this format has gained attention for applications like machine learning where small precision is sufficient in early stages of the algorithm and high throughput is desirable (assuming two operations on half numbers can be executed in the same time as one single precision operation). This extension is noteworthy because it lifts a limit imposed by `IeeeCC754`: All test vectors that come with `IeeeCC754` are only guaranteed to be valid for operands of at least 32 bits (i. e. for single and larger precisions). While most test vectors were also valid for 16 bit floating-point numbers, for some of the other vectors, there are mathematical reasons for this 32 bit limit (see [VCV01a; VCV01b]). Note that verification whether the test vectors contained in the current testsets constitute valid half precision test vectors is pending. However, all test vectors added in this thesis (as long as they are not precision dependent) represent valid test vectors (cf. Chapter 4). Furthermore, the parser discards test vectors whose operands or results are not representable in half format due to the exponent or significand being too large.

3.1.12 Analysis capabilities

Goal Use the information gained during the testing process to extract additional information about the underlying floating-point environment.

Reasoning `IeeeCC754` mainly concentrates on evaluating carefully chosen test vectors to verify correct rounding for difficult cases. Additionally, it uses the information gathered during the testing process to detect the underflow mode used in the current floating-point environment, see Section 2.2, [VCV01a], and [Cuy+02]. `IeeeCC754++` expands on this analysis and collects further data during the comparison step between a returned and the corresponding correct result. After the testing process has been finished, the results gained from analysing the test vectors are then written into the summary line(s) of the log file.

Realisation In addition to the underflow information that is already available in `IeeeCC754`, the following new analysis facilities have been added to `IeeeCC754++`:

- *Flush to zero detection*: IeeeCC754++ counts the number of operations whose correct result is a subnormal number (*tiny operations*, c_{tiny}). Out of these, the number of operations whose returned result is either zero (c_{zero}, indicating that it might have been flushed to zero) or the smallest normalised number ($c_{smallest}$, indicating that it was intended to set it to zero in FTZ mode, but was incorrectly rounded to a normalised number instead) is counted.

  After the completion of the testing process, these counts are evaluated: When $\frac{c_{zero}+c_{smallest}}{c_{tiny}} > \frac{2}{3}$, i. e. more than two thirds of the tiny operations returned an error, it is assumed that the current floating-point environment flushes subnormal values to zero, and a respective error is written to the summary. When $0 < \frac{c_{zero}+c_{smallest}}{c_{tiny}} \leqslant \frac{2}{3}$, a warning is logged that FTZ might have been used. Finally, $c_{tiny} > 0$ and $c_{zero} = c_{smallest} = 0$ means that the floating-point implementation under investigation supports subnormal values in an IEEE-conforming manner.

- *Deviations from correct results*: Whenever an operation is not correctly rounded and the correct result is either a subnormal or normalised number, the returned value differs from the expected one. When only a small number of leading digits of the significand are valid or when the exponent differs by more than 1, we assume that the implementation of the operation is flawed, and it should be further analysed in how far this implementation can be trusted. However, a rather small discrepancy between the returned and the correct result indicates that the implementation basically works, but that rounding is not as precise as it should be. In that case, the relative error ϵ will be small, typically in the order of a few ulps. IeeeCC754++ tries to determine which of these cases occurs more often in the current floating-point environment, and if values are only slightly off, it evaluates by how much the wrong results are off.

  In order to achieve this, IeeeCC754++ calculates the difference between the correct and the returned results for every test vector where the returned result differs from the correct one. If this difference is small enough (i. e. if the difference is smaller than a threshold of 8 ulps[1,2]), it is recorded to the log file together with other information about the test vector, cf. Section 3.3.2. If the difference is larger than this threshold, only the fact that a large deviation was found is logged.

  Note that the exact evaluation of the difference in ulps does not rely on the user environment's floating-point capabilities (which are under investigation during the test run), but rather exploits the binary representation of IEEE 754-2008 conforming floating-point numbers by interpreting the values

[1] This choice is rather generous: A relative error ϵ of 8 ulps accounts for the last three digits being wrong, depicting a quite inexact rounding accuracy.

[2] The actual threshold that is used by IeeeCC754++ is configurable, see Section 3.3.6.

as integers (see Section 1.2.1). Therefore, it suffices to use integer subtractions to compute the deviation in ulps between the correct and returned results.

Further analysis of the differences is left to other tools such as the evaluation framework, see Section 3.4.

- *Vector support*: To achieve better performance and floating-point throughput, modern floating-point hardware often supports vectorised floating-point operations, i. e. instead of performing a single floating-point operation, vector operand registers which are n times as wide as a regular floating-point register are first filled with n operands, and then one floating-point operation is executed that simultaneously works on the operand registers, thus performing n floating-point operations at the same time. In most cases, the same operation is executed for all operands contained in the vector registers (SIMD, Single Instruction Multiple Data).

  `IeeeCC754++` supports this type of floating-point operations by filling the vector registers with identical operands so that the same operation is executed n times. In addition to the normal checking process (i. e. checking the returned result and exceptions as usual), it is also tested whether all n results are identical. If this is not the case, a *vector error* is logged in order to indicate that there might be a problem in the FPU.

  Of course, it is also possible to execute a single operation on a vectorised FPU (i. e. on vector registers) by only using the first entry of the vector registers. If such a *scalar* operation is performed, `IeeeCC754++` offers facilities to check whether the unused entries of the result vector registers are left untouched, i. e. that they contain the same data as before the execution of the operation. If a value in one of these $n - 1$ entries was changed, a *scalar error* is logged.

  Note that the goal of this SIMD style vector support is not to provide exhaustive testing of all possible error variations. In particular, due to the way `IeeeCC754++` executes test vectors, generates results, and analyses them, possibly with the help of the evaluation framework, testing IEEE-conformity of a SIMD vector unit by executing different test vectors at the same time by placing operands in different vector entries is technically not viable. Furthermore, only basic checking of vector exceptions is done. This is mainly due to the fact that all current SIMD units that we are aware of do not feature exception registers containing separate values for the different entries in a SIMD vector. Often, only summarised exceptions are returned, i. e. every exception flag that is raised by one of the sub-operations is contained in the exception flags returned (cf. e. g. [INT17b]). However, in order to prepare for inconsistent exception flags being returned by SIMD operations, a *vector exception error* has been added to `IeeeCC754++`'s error flags and

the evaluation framework (although this error is currently not generated by any architecture port).

- *Zero values*: Especially when using FPUs not inside the processor (as e. g. accelerators, cf. Section 1.4.2), the FPU might not have been initialised correctly before computing on it. The results generated on an uninitialised FPU are meaningless and might be completely random (which nothing can be done about). Typically, however, the result of executing floating-point operations on an uninitialised FPU would be that all returned floating-point values be zero due to the operations returning default memory locations (which are usually zero). To ease the detection of this latter case, `IeeeCC754++` counts the number of zeroes returned as the result of a floating-point operation and warns the user of too many zeroes being generated by emitting two types of warnings: when either all values are zero (indicating that in fact no floating-point operations were performed by the FPU) or when more than 30% of the returned values are zero (which should only happen in abnormal cases).

- *`fma` error*: When performing an `fma` operation, IEEE 754-2008 mandates that only one final rounding be executed, i. e. that the intermediate result must be calculated with infinite precision. When a floating-point implementation does not perform "real" fused `fma`s, but instead simulates the operation by executing a multiplication followed by a separate addition, the results will be wrong for some test vectors. When `IeeeCC754++` detects wrong values as the result of an `fma` operation, an *fma* error is logged to warn the user that the `fma` might actually not be fused at all.

  Note that test vectors targeted at checking this property have been added to the `fma` testset, see Section 4.2.1.

These new additions are mainly targeted at the verbose mode in combination with the evaluation framework to ease the analysis and the evaluation of a larger number of test runs, e. g. when different compilers or compiler versions should be compared. However, all of the new analysis capabilities have been backported to the classic modes (see Sections 3.1.13 and 3.3.1) so they can be used to check e. g. vectorised FPUs or attain information about inconsistencies in `fma` or FTZ. As a consequence, `IeeeCC754++` writes extended information to the plain format log files compared to `IeeeCC754`, resulting in slightly larger log files.

3.1.13 Preserving backwards compatibility

Goal Keep features proven in `IeeeCC754`, but which are not in the primary focus of `IeeeCC754++`, working in `IeeeCC754++`, and backport relevant features that are compatible to `IeeeCC754`, or more precisely, to the classic modes.

Reasoning In order to provide a testing tool for IEEE 754-2008 conformity of current floating-point environments, `IeeeCC754++` enhances and updates `IeeeCC754` in a number of areas, such as support for floating-point features not in IEEE 754 or architectural changes to enable easy extensibility. However, care was taken to provide backwards compatibility where `IeeeCC754++` can benefit from a mature and well-tested tool as `IeeeCC754` is. This applies primarily to two areas: the support of `IeeeCC754`'s test vector format and the support for arbitrary precisions.

Realisation One of the main efforts of `IeeeCC754` development was the unification of test vectors in a variety of formats into one common range and precision independent format (Coonen) while providing routines to convert from Coonen format into UCB format that is closer to actual hardware implementations (and therefore range and precision dependent, i. e. every UCB test vector has fixed range and precision, cf. Section 2.2). `IeeeCC754++` makes use of the range and precision independent test vectors in Coonen format, although the syntax had to be slightly extended to support IEEE 754-2008 (for both Coonen and UCB format, see Section 3.2.1). However, this change has been done in a way that all test vectors that are valid in `IeeeCC754` are also valid in `IeeeCC754++`, i. e. the old syntax is still a valid subset of the new syntax. Moreover, test vectors from `IeeeCC754++`'s testsets that do not make use of the newly introduced IEEE 754-2008 features are still valid `IeeeCC754` input test vectors (in Coonen as well as UCB format). As a consequence, a user that developed test vectors for `IeeeCC754` can continue using these test vectors with `IeeeCC754++`.

Although `IeeeCC754++` has been extended with additional testing modes in order to support advanced analysis facilities (cf. Section 3.3), backwards compatibility has been preserved with regard to the three original `IeeeCC754` modes: testing with test vectors in either Coonen or UCB syntax and converting from Coonen to UCB format. This has been done in a way that the original syntax for calling `IeeeCC754` (see Listing C.3, page 335) can still be used with `IeeeCC754++` such that `IeeeCC754++` can behave (almost) identically to `IeeeCC754`. This approach enables `IeeeCC754++` to convert test vectors in Coonen format into UCB format (which is needed for the new modes added to `IeeeCC754++`, see Section 3.2) in arbitrary precisions. As a consequence, `IeeeCC754++`, which primarily focuses on standard precisions (namely, half, single, double, extended, and quadruple, cf. Table 1.1), can be used to check floating-point implementations with non-standard range and precision. Note that in this case (i. e. non-standard precisions), the use of `IeeeCC754++`'s extended analysis capabilities is not possible without further changes to `IeeeCC754++` itself. The newly introduced IEEE 754-2008 features are directly available in `IeeeCC754++`'s classic mode (for details, cf. Section 3.3.1).

3.1.14 Technical changes

Goal Update and restructure the IeeeCC754++ source code to support the new features explained above and enable future extensibility.

Reasoning The implementation of IeeeCC754 evolved as a series of student's programming projects at the University of Antwerp. It started off as a C program and was later ported to C++, at least partly. The evolutionary nature of the development resulted in a quite diverse code base. In order to support the new features presented in this thesis, especially the ability to add new architectures and ports (see Section 3.1.5), the new analysis facilities (see Section 3.1.12), and IEEE 754-2008 support (see Section 3.1.11), this code base needed to be cleaned up. C++03 [C++03] (which mainly was a bug-fix version of C++98 [C++98]) was chosen as the program language for IeeeCC754++ since parts of IeeeCC754 were already written in C++. Furthermore, using object orientation to implement the concepts of architectures and FPUs is an obvious choice (and IeeeCC754 already used inheritance for the implementation of the internal floating-point number representation). Using C++03 as programming language enables compilation of IeeeCC754++ with older compiler versions.

Realisation In order to clean up the code base and enable extensibility and portability, the following additional changes were introduced:

- New program features:
  - Support for comments in UCB format: Test vector files in Coonen format support comment lines, i. e. all lines starting with "!" (after some possible whitespace) are ignored by the parser. This functionality has been added to test vector files in UCB format.
  - To ease development and testing of new test vectors, a new option was added to the conversion mode -o in classic mode (cf. Section 3.3.1) to annotate the resulting UCB file with the source line number of the corresponding Coonen file. A detailed description can be found in Section 3.2.1.
- Parser changes:
  - Support for hexadecimal floating-point number input: In Coonen format, floating-point numbers are specified by entering a base number and manipulating this base with a range of modifiers. However, test vectors checking difficult cases for the elementary functions are only given as binary or hexadecimal numbers in the literature (cf. e. g. [LM01b] or [Mul+10]). Hexadecimal output was only allowed for integer numbers which are needed to encode test vectors for conversion between

floating-point and integer numbers. We extended the parser to support hexadecimal output for all operands for all operators.

- Support for operators with three operands has been added to the parser (see Section 3.1.11). Currently, this feature is only needed for the `fma` operation.

- Consistent use of `C++03`:
  - While I/O (input/output) to the system console had been rewritten to use `C++ iostreams`, file I/O still relied on `C` functions. Of course, the program worked as expected, but it was not possible to redirect I/O from the console to files and vice versa. Therefore, all I/O was changed to use `iostreams`.
  - Modern `C++` compilers favour the `C++ string` class over `C` strings (i. e. **`char`** arrays terminated by a 0 byte) and emit warnings when the latter are used. Furthermore, in `C++` constant strings defined in the source code are of type `string`. Occurrences of **`char`*** arrays have been rewritten to use `string`.
  - All occurrences of dynamic memory allocation were rewritten to use `C++`'s **`new`** and **`delete`** operators instead of `C`'s `malloc`/`calloc` and `free` functions.

- Necessary fixes:
  - Using Valgrind [VAL17], a number of memory leaks could be identified and fixed.
  - In the parser function `ReadALine()`, the definition of some local variables overshadowed global variables. As a consequence, in rare cases concerning NaN handling, parsing did not work correctly. The error has been fixed.

- Code cleanup:
  - The internal platform independent floating-point format of `IeeeCC754++` is encapsulated in the **`class`** `FP` (cf. Figure A.1 and Figure A.2 for an overview of `IeeeCC754++`'s class hierarchy). Due to the grown nature of `IeeeCC754`'s code base, its naming scheme was rather inconsistent, and several functions existed in slightly different versions (like e. g. two functions `isNaN()` and `isNan()` with different implementations to detect if the current `FP` number is an NaN). These inconsistencies have been removed, and in case of double implementations, the faster performing versions have been chosen.

- Overall, the code was transformed into a more readable code base by using consistent indentation and formatting and adding comments where necessary to improve legibility.
- Furthermore, data locality was exploited as far as possible. To achieve this, many of the variables that were formerly declared globally have been moved into the corresponding functions (where possible without changing semantics).
- Especially in the classes implementing `IeeeCC754++`'s internal data structures (like **`class`** `FP`, see above, but also in the `Error` classes, see Figure A.2), the code relies on integers having a fixed (and known) size. Since `C/C++` do not require exact sizes for integers, but only certain rules like minimum sizes and relations between the different integer datatypes like **`int`** or **`long`**, the size of integers is user environment (and compiler) dependent. Therefore, it is necessary to know the exact specifications of the currently used user environment at compile time. All relevant occurrences of integer types where the exact size (and/or signedness) matter have been rewritten to use the standard header `cinttypes` (or `inttypes.h` as a fallback) which provide datatypes such as `int32_t`, `uint64_t` etc. The availability of these headers and alternative handling methods are detected and implemented via Autoconf, see Appendix A.

3.2 Input and output

`IeeeCC754++` follows `IeeeCC754`'s philosophy and employs an extended driver program to test the floating-point environment. Differentiation for the different use cases described in Section 3.1 is mainly done via two complementary means: on the one hand using specialised output modes (see Section 3.3) and on the other hand calling architecture ports which denote `IeeeCC754++`'s implementations for a given hardware or software FPU (cf. Chapter 5).

In this section, we briefly describe the input formats used to feed test vectors into `IeeeCC754++` and the various output formats employed by `IeeeCC754++` in conjunction with the corresponding testing modes.

3.2.1 Test vector formats

Tests in `IeeeCC754++` are executed in the following way: A test vector is read from file and encoded into target format. The desired operation in the requested rounding mode as described in the test vector is executed with the encoded operands. The returned result is then compared to the correct result (which is also stored in the test vector), and it is checked whether the correct exceptions were flagged. For more details on the testing process, see Chapter 2.2.

This above description is actually slightly simplified. There exist two different input file formats for describing test vectors, the Coonen format which features range- and precision-independent descriptions of the test vectors, and the UCB format in which each test vector is rendered in hexadecimal notation for exactly one precision and one rounding mode. `IeeeCC754++` accepts test vectors in both formats. When UCB input is used, one test vector per input line is converted from hexadecimal into binary machine format, possibly taking endian issues into account, and is executed afterwards. An input line in Coonen input is treated differently: The input vector is first translated into up to ten vectors in UCB format, one for each of five possible rounding modes and taking commutativity into account for addition, multiplication, and `fma`[3]. Afterwards, it is treated exactly as if read from an input file in UCB format.

`IeeeCC754++`'s test vectors are stored in the most general format, i. e. in Coonen format. To generate input files in UCB format, `IeeeCC754++` uses the conversion mode (contained in the classic mode) that writes the internally generated UCB test vectors into a file. For an overview of the input and conversion facilities as well as the testing process, see Figure 2.1, page 42.

In the following, we give short descriptions of the changes applied to the test vector file formats in order to support the features introduced with `IeeeCC754++`, together with the resulting grammars in BNF (Backus-Naur Form) [Knu64]. Note that a detailed description of the syntax, including grammars of the original syntax, can be found in [VCV01a] and [VCV01b].

Coonen input files

The Coonen test vector syntax has been extended to allow for the following new features in `IeeeCC754++`:

- Support for roundTiesToAway: The parser now accepts the character "`~`" as a valid rounding mode and generates UCB test vectors for roundTiesToAway mode accordingly.
- We added all floating-point operators recommended (but not required) by IEEE 754-2008 (cf. e. g. Section 3.1.11) to the test vector syntax and the parser. A detailed list of the operators can be found in Table 4.6; the corresponding names are given in the Coonen format grammar, see Listing 3.1.
- The half precision format can be chosen by using the newly introduced precision specifier "`h`".
- Finally, "`W`" (as in "`Wuppertal`" has been added as version specifier to denote test vectors that were added within this thesis.

[3]Note that for `fma`, commutativity can only be exploited in the multiplication due to the fixed ordering of operands.

The extended grammar in BNF form is shown in Listing 3.1.

```
<line> ::= <testvector> | <comment>
<testvector> ::= <version><operation> <prec> <rounding> <fp> <fp> {<fp>}
    <exceptions> <fp>
<version> ::= <digit> | "H" | "A" | "W"
<operation> ::= "+" | "-" | "*" | "/" | "%" | "S" | "fma" | <conv> | <elem>
<conv> ::= "r" | "c" | "i" | "d2b" | "b2d" | <intconv>
<intconv> ::= "ci" | "ri" | "cu" | "ru" | "cI" | "rI" | "cU" | "rU"
<elem> ::= <pow> <trig> <hyper> <exp> <log> <misc>
<pow> ::= "cbrt" | "rootn" | "pow" | "pown" | "powr"
<trig> ::= "sin" | "cos" | "tan" | "sinpi" | "cospi" | "atanpi" | "atan2pi"
         | "asin" | "acos" | "atan" | "atan2"
<hyper> ::= "sinh" | "cosh" | "tanh" | "asinh" | "acosh" | "atanh"
<exp> ::= "exp" | "expm1" | "exp2" | "exp2m1" | "exp10" | "exp10m1"
<log> ::= "log" | "log2" | "log10" | "logp1" | "log2p1" | "log10p1"
<misc> ::= "hypot" | "rsqrt" | "comp" | "erf" | "erfc" | "gam" | "lgam"
<prec> ::= {"e" | "o" | "h{ieee}" | "s{ieee}" | "d{ieee}" | "l{ieee}"
         | "q{ieee}" | "m{ieee}"}
<rounding> ::= "ALL" | "0" | "<" | ">" | "=" | "~" | "0<" | "0>"
             | "=<" | "=>" | "=0>" | "=0<" | "0<~" | "0>~"
             | "=<~" | "=>~" | "=0>~" | "=0<~" | "UN"
<exceptions>::= "OK" | "x" | "xo" | "xu" | "xv" | "xw" | "i" | "z"
<fp>::= {<sign>}<root>{<suffix>}* | <decimal> | <integer>
<integer>::= {"?"}"0x"{<hex>}+
<decimal>::= <sign> {<hex>}+{"_"{<digit>}+"&"} "E"<sign>{<digit>}+
<sign>::= "+" | "-"
<root>::= "Q" | "H" | "T" | {<digit>}+
<suffix>::= "p"<literal> | "m"<literal> | "i"<spec> | "d"<spec> | "u"<digit>
<spec>::= <digit> | (<pos>)<digit>
<pos>::= <literal>{"+"<digit>} | <literal>{"-"<digit>}
<literal>:: = <digit> | "t" | "h" | "B" | "B"<digit> | "u" | "C"
<hex>::= <digit> | "a" | "b" | "c" | "d" | "e" | "f"
<digit>::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<comment> ::= "!" .*
```

Listing 3.1: *Grammar in BNF form for test vector files in Coonen extended syntax.*

UCB input files

The changes introduced to the UCB test vector syntax are slightly more extensive than in the Coonen case, although most of the changes reflect the features that were already addressed with the extensions to the Coonen syntax:

- Support for roundTiesToAway: The parser now accepts the character "`a`" as a valid rounding mode in UCB files.
- The list of operators added to the UCB syntax is identical to that in the Coonen case.
- The half precision format can be specified by "`h`", and test vectors added within this thesis can be marked with "`W`".

- The new extended UCB syntax allows for floating-point numbers to be given in hexadecimal format. In the older syntax, this was only possible for integer values (i. e. only in the case of conversions from and to integer format).
- Finally, comment lines have been introduced. Similar to the Coonen syntax (which borrows the comment character "`!`" from `FORTRAN`), every line starting with "`!`" is ignored.

The extended grammar in BNF form is shown in Listing 3.2.

```
<line> ::= <testvector> | <comment>
<testvector> ::= <op><format> <rounding> <compare> <exceptions>
    <fp> <fp> {<fp>} <fp>
<op> ::= "add" | "sub" | "mul" | "div" | "rem" | "sqrt" | "fma"
       | <conv> | <elem>
<conv> ::= "r" | c" | "i" | "d2b" | "b2d" | <intconv>
<intconv> ::= "ci" | "ri" | "cu" | "ru" | "cI" | "rI" | "cU" | "rU"
<elem> ::= <pow> | <trig> | <hyper> | <exp> | <log> | <misc>
<pow> ::= "cbrt" | "rootn" | "pow" | "pown" | "powr"
<trig> ::= "sin" | "cos" | "tan" | "sinpi" | "cospi" | "atanpi" | "atan2pi"
         | "asin" | "acos" | "atan" | "atan2"
<hyper> ::= "sinh" | "cosh" | "tanh" | "asinh" | "acosh" | "atanh"
<exp> ::= "exp" | "expm1" | "exp2" | "exp2m1" | "exp10" | "exp10m1"
<log> ::= "log" | "log2" | "log10" | "logp1" | "log2p1" | "log10p1"
<misc> ::= "hypot" | "rsqrt" | "comp" | "erf" | "erfc" | "gam" | "lgam"
<format> ::= "h" | "s" | "d" | "l" | "q" | <exp> <hidden> <prec>
<exp> ::= <digit>+
<hidden> ::= "0" | "1"
<prec> ::= <digit>+
<rounding> ::= "n" | "p" | "m" | "z" | "a"
<compare> ::= "eq" | "uo"
<exceptions> ::= "-" | "x" | "xo" | "xu" | "xa" | "xb" | "v" | "d"
<fp> ::= {<hex><hex><hex><hex><hex><hex><hex><hex>}* {<hex><hex><hex><hex>}+
<hex> ::= <digit> | "a" | "b" | "c" | "d" | "e" | "f"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<comment> ::= "!" .*
```

Listing 3.2: *Grammar in BNF form for test vector files in UCB syntax.*

3.2.2 Output formats

The output formats used in `IeeeCC754++` have already been mentioned in Section 3.1.10. Since these specialised output formats are closely related to the corresponding testing modes, they are discussed in the next section.

3.3 Testing modes

In this section, we discuss the testing modes that have been implemented in `IeeeCC754++` to test floating-point environments according to the various use cases described in Section 3.1. Before introducing these modes, we need to define the corresponding terms:

Definition 3.4. An *input mode* is a collection of test vectors in either Coonen or UCB format that are fed into `IeeeCC754++` in order to assess the IEEE-conformity of a user environment, while an *output mode* depicts an output format containing the information gathered during the testing and analysis stage, specifically process for the target use case.

A *testing mode* then consists of the combination of an input mode together with an output mode. Additionally, it may optionally include a specific architecture port. ◊

Note that for all testing modes added to `IeeeCC754++`, the UCB format has been chosen as input mode. For a detailed discussion, cf. Section 3.1.10. An overview of all testing modes and their relation to input and output formats can be found in Figure 3.1.

3.3.1 Classic mode

We start the discussion of testing modes with the classic mode which actually consists of three modes: `IeeeCC754++` includes all three modes that were already available in `IeeeCC754`. This approach provides backwards compatibility and provides the only means to convert test vectors from Coonen to UCB format. The syntax for these modes is as follows:

```
> IeeeCC754++ <MODE> <OPTIONS> [-f <LOGFILE>] <INFILE>
```

and

```
> IeeeCC754++ -o <OUTFILE> [-a] <OPTIONS> <INFILE>
```

where `<LOGFILE>` is an optional name for the log file (default is `ieee.log`), `<INFILE>` is the input file in Coonen or UCB format, and `<OPTIONS>` are additional options to influence the testing phase as described in Listing C.3, page 335.

`<MODE>` determines the testing mode:

- *Coonen mode* is called by specifying `-c` as `<MODE>`. In this mode, test vector files are expected to be in Coonen format, and the output format of the text file `<LOGFILE>` is `IeeeCC754`'s description of errors that were encountered during the testing process (see below).

  Since input is read in the range- and precision-independent Coonen format, the exact floating-point format to be tested must be specified, e. g. by using the options `-h`, `-s`, `-d`, `-l`, or `-q` for numbers in half, single, double, extended, or quadruple format.

- When `-u` is used, `IeeeCC754++` works in *UCB mode* which is almost identical to Coonen mode. The main difference is the expected test vector format: input is expected to be in UCB format, and consequently these (hexadecimal) test vectors are immediately converted to machine format. The output format

however is the same as in Coonen mode, namely the plain format (see below). Additional floating-point format specifications are not necessary as the range and precision information is encoded into the test vectors. Therefore, it is possible to test multiple floating-point formats during one `IeeeCC754++` run (which is not possible in Coonen mode).

- The mode switch `-o` selects output or *conversion mode*. After reading test vectors in Coonen format, the corresponding test vectors in UCB format are generated and written into the output file. It should be noted that the syntax is slightly different in this case as the file name of the output file must be specified directly after the option `-o`.

  Like in Coonen mode, the target floating-point format must be explicitly given via the respective command line options.

The output format of the conversion mode is a test vector file in UCB format, whereas in Coonen and UCB mode, a *plain format* log file is generated containing the following information: the command line that `IeeeCC754++` was called with, a summary including the total number of vectors tested as well as the number of errors, warnings, and skipped vectors processed, and the underflow mode that was detected (if it could be detected). Furthermore, for every error that was encountered, detailed information about the operands as well as the correct and the computed result are printed, including operation and rounding mode used and description of every error that was detected. Listing 3.3 shows (heavily shortened) example output that was produced by running `IeeeCC754++` in UCB mode on a large set of test vectors in single format.

```
Testrun: ./IeeeCC754++ -u -f IeeeCC754-UCB-output.log alls

...

Error Line 33881: inexact flag not returned
Operation: b2d
Round to nearest (ties to even)
Operand 1: 4604d000
Operand 2: 00000000
Flags expected: x
Flags returned:
Correct result:  +8E+3
Returned result: +8E+3

...

Error Line 33949: inexact flag not returned
Error Line 33949: different decimal representation
Operation: b2d
Round to nearest (ties to even)
Operand 1: 5fad78ec
Operand 2: 00000000
Flags expected: x
Flags returned:
Correct result:  +3E+19
```

```
Returned result: +2E+19

...

Summary:
--------
Implementation signals underflow in case the result
(1) is tiny after rounding and
(2) raises the inexact exception
('v' - underflow)
Errors:   3930/24573
Warnings: 0/24573
Skipped:  0/24573
```

Listing 3.3: *Example output in plain format generated with `IeeeCC754++ -u alls`. The operation in the shown examples is conversion from binary to decimal (`b2d`), so the results are printed in decimal notation.*

To ease development and testing of new test vectors, a new option was added to the conversion mode `-o` in classic mode (cf. Section 3.3.1): When `-a` is added to the command line, the output UCB file will be annotated with the source line numbers of the Coonen file. More precisely, before writing a converted test vector in UCB format to the output file, a comment line including the line number of the (unconverted) test vector in the Coonen input file is written to the UCB file. Listing 3.5 shows example UCB output in single precision generated from the Coonen input file shown in Listing 3.4. Please note that line 1 is ignored as it is a comment line and that the test vector in line 2 is valid for all rounding modes ("`ALL`" in column 2). Therefore, ten test vectors in UCB mode are generated (two vectors for each of the five rounding modes to check commutativity), and each test vector line is preceded by a comment line showing that it was generated from line 2 of the input file.

```
!!!! first some easy cases
2+       ALL   1   2    OK   4
```

Listing 3.4: *coonen.in: Example Coonen test vector file.*

```
! ./IeeeCC754++ -o ucb.out -a -s ./coonen.in
! line 2
adds n eq - 3f800000 40000000 40400000
adds n eq - 40000000 3f800000 40400000
! line 2
adds z eq - 3f800000 40000000 40400000
adds z eq - 40000000 3f800000 40400000
! line 2
adds p eq - 3f800000 40000000 40400000
adds p eq - 40000000 3f800000 40400000
! line 2
adds m eq - 3f800000 40000000 40400000
adds m eq - 40000000 3f800000 40400000
! line 2
adds a eq - 3f800000 40000000 40400000
```

```
adds a eq - 40000000 3f800000 40400000
```

Listing 3.5: *Annotated UCB test vector file in single precision generated with* `IeeeCC754++ -o ucb.out -a -s coonen.in`.

3.3.2 Verbose mode

The verbose mode has been developed as an all-purpose mode for a wide range of testing applications. To achieve this goal, it features a specially designed output format that provides an extensive overview of the testing process while at the same time being machine parsable and therefore enabling further evaluation of the test results. When paired with specific architecture ports, it serves as the main platform for testing specific hardware parts or FPUs. The default mode is basically the verbose mode together with the `default` architecture, see Section 3.3.3.

The verbose mode takes an input file in UCB format, executes the specified operations in the target floating-point environment, and writes information about the test vectors into the output file in the newly developed *verbose output format.*. The syntax is a variant of the classic mode syntax:

```
> IeeeCC754++ -v<MODE> <INFILE> [-f <LOGFILE>] [<OPTIONS>]
```

with `<INFILE>` in UCB format and the possibility to change the default name for `<LOGFILE>` (which is `<INFILE>.log`). `<OPTIONS>` describes further general options (see Section 3.3.6) or options that depend on the architecture port being used.

The main difference between the verbose mode and the other modes is the format of the log file. `IeeeCC754`'s output format describing all errors encountered during the testing process is not particularly well-suited for parsing. In order to achieve better machine-readability while at the same time keeping human legibility, we developed the verbose output format whose grammar is described in Listing 3.6:

```
<line> ::= <pre> <input> "=>" <output> <errors>
<pre> ::= <out spec> <lineno>
<input> ::= <prec> <operation> <rounding> <fp> <fp> [<fp>] <fp> [<exceptions>]
<output> ::= <errchar> <fp> [<exceptions>]
<errors> ::= [<vectorerrorspec>] [<errorspec>]
<out spec> ::= "[" <outputspecifier> "]"
<outputspecifier> ::= "io" | "ix" | "cc" | "i_" | "_o"
<lineno> ::= "l."<integer>
<prec> ::= "h" | "s" | "d" | "l" | "q"
<operation> ::= "add" | "sub" | "mul" | "div" | "rem" | "sqrt" | "fma"
              | "rt" | "ct" | "i" | "b2d" | "d2b"
              | "ri" | "ru" | "rI" | "rU" | "ci" | "cu" | "cI" | "cU"
              | "cbrt" | "rootn" | "pow" | "pown" | "powr"
              | "sin" | "cos" | "tan" | "sinpi" | "cospi" | "atanpi" | "atan2pi"
              | "asin" | "acos" | "atan" | "atan2"
              | "sinh" | "cosh" | "tanh" | "asinh" | "acosh" | "atanh"
              | "exp" | "expm1" | "exp2" | "exp2m1" | "exp10" | "exp10m1"
              | "log" | "log2" | "log10" | "logp1" | "log2p1" | "log10p1"
              | "hypot" | "rsqrt" | "comp" | "erf" | "erfc" | "gam" | "lgam"
```

```
<rounding> ::= "n" | "p" | "m" | "z" | "a"
<exceptions> ::= ["i"]["z"]["x"]["o"]["u"]["v"]["w"]
<errchar> ::= "+" | "-"
<vectorerrorspec> ::= [<vectorerror>][<vectorexceptionerror>][<scalarerror>]
<vectorerror> ::= "*"
<vectorexceptionerror> ::= "/"
<scalarerror> ::= "#"
<errorspec> ::= <errordelim> <errordesc> <integer>
<errordelim> ::= "|"
<errordesc> ::= {<char> | ">"}*

<summary> ::= <preamble> <counts> <underflow> <ftz> [<zeroes>]
<preamble> ::= "[sum] total/err/warn/skip"
<counts> ::= <integer>"/"<integer>"/"<integer>"/"<integer>
<underflow> ::= "m" | "u" | "v" | "w" | "ov" | "ow" | "ovw" | "nv" |"nw" | "nvw"
    | "i" | "k"
<ftz> ::= {"[ftz]" | "ftz?"} <integer>"/"<integer>"/"<integer> | "[noftz]" |
    "[FTZ]" | "[FTS]"
<zeroes> ::= "[ZEROES]" | "[ZEROES30]"

<integer> ::= {<digit>}+
<fp> ::= {<hex><hex><hex><hex>}+
<hex> ::= <digit> | "a" | "b" | "c" | "d" | "e" | "f"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<char> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
         | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t"
         | "u" | "v" | "w" | "x" | "y" | "z"
```

Listing 3.6: *Syntax for the verbose output format in Backus-Naur form.*

For every test vector, one line of output is generated. Every line consists of up to four sections: a *general part* including the source line number, an *input part* which shows the operands, the (correct) result, and expected exception flags (if any), an *output part* which shows the returned result and exceptions, and an *error part* which shows more detailed information on the type of error if one occurred for the corresponding test vector.

- The general part shows information which helps to trace back what operation was performed from which input line. It starts with a four character long descriptor showing the chosen output mode (see below) and the source line number. It then lists the precision of the operands and the operation and rounding mode under consideration.

- The input part consists of hexadecimal representations of the operands and the correct results. Additionally, the exceptions flags expected to be returned are listed.

- If input and output parts are present, the delimiter "`=>`" is printed.

- The output part starts with a "success character", i. e. either a "`+`" if a correct result was returned or a "`-`" if the returned result was not fully IEEE-conforming, thus making it easy to calculate success and failure rates.

Furthermore, the computed result and the returned exception flags are shown.

- Finally, if an error occurred (i. e. for every line that includes a "`-`" as success specifier), additional information is printed, starting with the delimiter "`|`" and followed by a sequence of characters describing the nature of the error that occurred. These characters are shorthands for the errors used in `IeeeCC754++`'s classic modes, for details see Table C.2, page 338. The error part concludes with the difference between the correct and the returned result in ulps iff the returned result is a number (either normalised or subnormal), the exponents do not differ by more than one, and the difference does not exceed a given threshold. This way, very large differences are not recorded (because the result is wrong to an extent that the difference to the correct result in ulps is meaningless), and it is possible to evaluate by how much the returned results differs from the expected result. For a more detailed discussion, see Section 3.1.12.

Additionally, a summary line starting with `[sum]` is printed showing the number of vectors tested, how many out of those produced errors, warnings, or were skipped, and the detected underflow mode. Furthermore, if too many of the generated results were 0 (which might indicate an error like missing driver initialisation for GPUs, cf. e. g. Section 5.5.1), a warning is emitted. `IeeeCC754++` also tries to detect if FTZ is used and generates corresponding output: `[noftz]` indicates that FTZ was not detected, `[FTZ]` and `[FTS]` denote that `IeeeCC754++` was started in "flush to zero" and "flush to signed zero" mode via the command line options `--ftz` and `--ftzsigned` (cf. Section 3.3.6), respectively, and `[ftz]` and `[ftz?]` report that FTZ was (or, in the second case, was maybe) switched on.

In order to achieve a high degree of flexibility, `IeeeCC754++` provides the following variants of the verbose output format, selectable via the `<MODE>` specifier:

- `-vio`, also `-v`: This is the default variant of the verbose mode. It uses all four output sections to show a comprehensive overview of testing results, beginning with a prefix of "`[io]`". In this format, all information included in the classic modes is also available, albeit in a condensed format that is efficiently parsable. In addition to that, information about all test vectors is kept, especially about those which returned IEEE-conforming results.

  Listing 3.7 shows heavily shortened example output, containing one output line for a test vector where no errors occurred and the two test vectors also shown in Listing 3.3.

```
[io] l.3 s add n 3f800000 3f800000 40000000  => + 40000000
...
[io] l.33881 s b2d n 4604d000 00000000 +8E+3 x => - +8E+3   |  j 0
```

```
...
[io] l.33949 s b2d n 5fad78ec 00000000 +3E+19 x => - +2E+19   |  jd 0
...
[sum] total/err/warn/skip 24573/3930/0/0 v [noftz]
```

Listing 3.7: *Example verbose format output generated with* `IeeeCC754++ -v alls`*.*

- `-vix` calls a variant of `-vio` showing basically the same output, with one notable exception: Test vectors for which only errors related to exceptions are reported (i. e. for which expected exceptions were not raised or for which additional exceptions were flagged) are not counted as errors. This means that a "`+`" is shown as success specifier, and the error counter is not increased. Lines written in `-vix` mode are prefixed with "`[ix]`".

  To still be able to retrieve information about which errors happened, the error part of a `-vio` output line is still printed. Listing 3.8 shows the differences to `-vio` mode.

```
[ix] l.3 s add n 3f800000 3f800000 40000000  => + 40000000
...
[ix] l.33881 s b2d n 4604d000 00000000 +8E+3 x => + +8E+3   |  j 0
...
[ix] l.33949 s b2d n 5fad78ec 00000000 +3E+19 x => - +2E+19   |  jd 0
...
[sum] total/err/warn/skip 24573/175/0/0 v [noftz]
```

Listing 3.8: *Example verbose format output generated with* `IeeeCC754++ -vix alls`*.*

- `-vcc` is a technical variant developed in order to enable fast comparison of results to the known correct result. Although the same testing process is performed as in the other verbose modes, the known correct result and exceptions as stored in the source test vector and printed in the input section are mirrored to the output section. In this variant, the only part of the output file depending on the testing process is the summary line where underflow mode, FTZ, and zeroes are shown as detected.

  Listing 3.9 shows example output. Note that to ease using commands like `diff` on the command line, the prefix is identical to the one in `-vio` mode.

```
[io] l.3 s add n 3f800000 3f800000 40000000  => + 40000000
...
[io] l.33881 s b2d n 4604d000 00000000 +8E+3 x => + +8E+3 x
...
```

```
[io] l.33949 s b2d n 5fad78ec 00000000 +3E+19 x => + +3E+19 x
...
[sum] total/err/warn/skip 24573/0/0/0 v [noftz]
```

Listing 3.9: *Example verbose format output generated with* `IeeeCC754++ -vcc alls`.

- Finally, there exist two shorter output variants which include either only the input part (`-vin`) or the output part `-vout` of a "full" verbose output. Listings 3.10 and 3.11 show the corresponding example lines.

```
[i_] l.3 s add n 3f800000 3f800000 40000000
...
[i_] l.33881 s b2d n 4604d000 00000000 +8E+3 x  |  j 0
...
[i_] l.33949 s b2d n 5fad78ec 00000000 +3E+19 x  |  jd 0
...
[sum] total/err/warn/skip 24573/3930/0/0 v [noftz]
```

Listing 3.10: *Example verbose format output generated with* `IeeeCC754++ -vin alls`.

```
[_o] l.3 s add n + 40000000
...
[_o] l.33881 s b2d n - +8E+3   |  j 0
...
[_o] l.33949 s b2d n - +2E+19   |  jd 0
...
[sum] total/err/warn/skip 24573/3930/0/0 v [noftz]
```

Listing 3.11: *Example verbose format output generated with* `IeeeCC754++ -vout alls`.

3.3.3 Default mode

The default mode is the only testing mode that combines an output mode (the verbose output format) with an architecture port: It relies on `IeeeCC754++` having been built with the default architecture, i. e. an implementation of the floating-point operators that uses only the native operators supplied by `C/C++`. The goal is to mimic a "standard" user's computing environment as far as possible. Therefore, the choice of the exact execution location of the floating-point commands is left to the compiler, possibly influenced by optimisation and other compiler

options supplied at build time. This approach makes it possible to test the native environment in an easy manner and to evaluate the influence of compiler options on the numerical performance immediately.

To employ the default mode, `IeeeCC754++` must be built using the default architecture, and the resulting executable is executed in verbose mode. Details on the build process and how custom compiler options can be incorporated are given in Appendix A. Additionally, the actual implementation of the operators is discussed in Section 5.1.1.

As the calling conventions are identical to those of the verbose mode, all different output formats of the verbose mode can be used in the default mode.

3.3.4 Distributed computing modes

`IeeeCC754++` includes two modes aimed specifically at distributed computing. In this section, we give a short overview of these modes including the calling conventions for completeness reasons. A short overview of distributed computing as used in this thesis is given in Section 3.1.4.

Checksum mode

The checksum mode's goal is to provide an easy means to prevent execution of a numerical program on a user environment where the floating-point behaviour is different from a known "local" user environment. It works in a manner similar to the UCB mode (cf. the classic mode, Section 3.3.1), but features a condensed binary output format that keeps almost all information of the classic mode's plain output format, but uses only exactly 16 bytes per error plus an additional 37 bytes for the summary header, cf. Tables 3.1 and 3.2 for the exact format. This approach drastically decreases the size of the log file especially in case of many errors and thus increases comparison performance. Note that an early version of this mode is discussed in more detail in [HF06].

The syntax of the checksum mode is as follows:

```
> IeeeCC754++ -s <INFILE> [-f <LOGFILE>] [<OPTIONS>]
```

with the default name for `<LOGFILE>` again being `<INFILE>.log` and `<INFILE>` being in UCB format. Possible options are described in Section 3.3.6.

Similar to the difference between `-vio` and `-vix` in verbose mode, a variant of the checksum mode called *dropped checksum* mode exists in which test vectors for which only errors related to exception flags were detected are not counted as errors. The syntax is almost identical to checksum mode:

```
> IeeeCC754++ -d <INFILE> [-f <LOGFILE>] [<OPTIONS>]
```

In case not only the comparison between checksums generated on different nodes is desired but also insight into the actual nature of the errors encountered, we developed a small utility called `decode` that takes a checksum mode log file as

Encoded information	Number of bytes
Endianess	1
Number of errors	4
Number of warnings	4
Number of skipped vectors	4
Total number of vectors	4
Summary (underflow mode)	4
Number of tiny operations	4
Number of FTZ operations	4
Number of FTZ errors	4
Number of zero values	4
Total number of bytes	37

Table 3.1: *Checksum mode header.*

Encoded information	Number of bytes
Precision	1
Operation	1
Rounding mode	1
Errors	4
Expected exceptions	1
Returned exceptions	1
Errorlevel	1
Deviation in ulps	2
Source line number	4
Total number of bytes	16

Table 3.2: *Checksum mode error. For every error that is encountered during the testing process, one entry with the information listed here is written to the logfile.*

input and generates an output file in (almost) the classic mode plain format. The only difference lies in the fact that in checksum mode, the operands and results are not stored and therefore can not be recovered and printed in the log file.

The decoding utility is called as follows:

```
> decode [-s|-d] <INFILE> [-f <OUTFILE>]
```

with `-s` and `-d` specifying the decoding mode. If the name of `<OUTFILE>` is not explicitly specified via the `-f` option, the default name `<OUTPUT>.decoded` is used. If `<INFILE>` was encoded in checksum mode (`IeeeCC754++ -s`), `decode -s` recovers all information as stored in the binary checksum, whereas `decode -d` behaves exactly as if `<INFILE>` had been encoded in dropped checksum mode, i. e. it drops all errors only related to exceptions. When `<INFILE>` was encoded in

dropped checksum mode, both decoding modes produce identical output.

Listings 3.12 and 3.13 show examples of decoded log files which were produced by `IeeeCC754++ -s` and `IeeeCC754++ -d` respectively.

```
Error Line 33881: inexact flag not returned
Operation: b2d
Round to nearest (ties to even)
Flags expected: x
Flags returned:

...

Error Line 33949: inexact flag not returned
Error Line 33949: different decimal representation
Operation: b2d
Round to nearest (ties to even)
Flags expected: x
Flags returned:

...

Summary:
--------
Implementation signals underflow in case the result
(1) is tiny after rounding and
(2) raises the inexact exception
('v' - underflow)
Errors:   3930/24573
Warnings: 0/24573
Skipped:  0/24573
```

Listing 3.12: *Example checksum mode plain format output generated with `IeeeCC754++ -s alls ; decode alls.log`*

```
Error Line 33949: different decimal representation
Operation: b2d
Round to nearest (ties to even)
Flags expected:
Flags returned:

...

Summary:
--------
Implementation signals underflow in case the result
(1) is tiny after rounding and
(2) raises the inexact exception
('v' - underflow)
Errors:   175/24573
Warnings: 0/24573
Skipped:  0/24573
```

Listing 3.13: *Example dropped checksum mode plain format output generated with `IeeeCC754++ -d alls ; decode alls.log`*

Note that the output in Listing 3.13 could have also been produced with the following two commands:

```
> IeeeCC754++ -s alls
> decode -d alls.log
```

Fingerprint mode

If `IeeeCC754++` reports a large number of errors for a given test set, the checksum files generated in checksum mode can still be of substantial size. Under certain circumstances, it can be advantageous to work with a further reduction of the checksums, e. g. when checksum information should be stored in some database. To support this use case, `IeeeCC754++` features the *fingerprint* mode that generates a hash digest of the recorded error information and stores it (in binary format) together with the checksum mode header. The size of this fingerprint is then 37 bytes for the header plus the size of the chosen digest mode (typically 128 or 256 bit), i. e. it is in the range of $\mathcal{O}(50)$ bytes.

The syntax is very similar to that of the checksum mode:

```
> IeeeCC754++ -h[<DIGEST>] <INFILE> [-f <LOGFILE>] [<OPTIONS>]
> IeeeCC754++ -m           <INFILE> [-f <LOGFILE>] [<OPTIONS>]
```

with `<INFILE>`, `<LOGFILE>`, and `<OPTIONS>` as in checksum mode. `-h` stands for *hash digest*; the default digest is SHA1. It is possible to select a different hash digest by specifying `<DIGEST>`, e. g. using `-hsha512` generates an SHA512 digest. Consequently, `-hsha1` is identical to `-h`. `-m` is a shortcut for `-hmd5` and selects the MD5 digest.

To generate a readable version of the hash digest in hexadecimal format and to retrieve the information encoded in the summary header, `decode` can be used with the following syntax:

```
> decode -h[<DIGEST>] <INFILE> [-f <OUTFILE>]
> decode -m           <INFILE> [-f <OUTFILE>]
```

It is important to note that `decode` has no knowledge of the hash digest that was used to generate the fingerprint. It will therefore transform the digest into hexadecimal format assuming that the digest specified on the command line was the one used during the encoding process. If the digest length of the selected digest is (by chance) identical to that of the digest used in `IeeeCC754++ -h` mode, `decode` will print the right value, albeit with a wrong digest name. If digest lengths differ, appropriate warnings are printed.

Listing 3.14 shows example output for a fingerprint encoded with SHA1 digest.

```
Summary:
--------
Implementation signals underflow in case the result
(1) is tiny after rounding and
(2) raises the inexact exception
('v' - underflow)
Errors:   3930/24573
Warnings: 0/24573
Skipped:  0/24573
```

```
SHA1 Hash digest:
f584bd2c488d687b36bdb52416f564a08c3f7998
```

Listing 3.14: *Example fingerprint mode output generated with* `IeeeCC754++ -hsha1 alls ; decode -hsha1 alls.log`

3.3.5 Miscellaneous modes

In addition to the modes listed above, there exist two modes serving more technical purposes:

```
> IeeeCC754++ -q <INFILE> [<OPTIONS>]
> IeeeCC754++ -t
```

`-q` calls the *quiet mode* that does not generate output at all; it can be used to evaluate the difference in execution speed when applying different levels of optimisation during compilation. If a log file would be generated, the performance would be largely I/O bound. By dropping all output, the tests are executed as usual, but performance is not impeded by generating potentially large output files.

When called with `-t`, `IeeeCC754++` prints technical information about itself to the command line such as a version number, the architecture port it was compiled with and the compiler used, the subversion revision it was compiled from, and the Autotools versions used to setup the build system. Listing 3.15 shows example output.

```
Calling main implementation (no FPU kernel requested).
IeeeCC754++ driver executable v0.9.5-dev.
64 bit version compiled for architecture "default" on x86_64 with gcc (4.8.5).
Built against revision r947
autotools version:
  autoconf: autoconf (GNU Autoconf) 2.69
  automake: automake (GNU automake) 1.13
  m4:       m4 (GNU M4) 1.4.17
```

Listing 3.15: *Output of* `IeeeCC754++ -t`

3.3.6 Common command line options

So far, we have discussed the different testing modes supported by `IeeeCC754++` which can be called with the following syntax:

```
> IeeeCC754++ -<MODE> <INFILE> [-f <LOGFILE>] [<OPTIONS>]
```

Via the `[<OPTIONS>]` parameters that are available in all extended modes, i. e. in all modes with the exception of the classic modes from Section 3.3.1, several additional options can be used that either provide the user with information on `IeeeCC754++` usage or that add further functionality common to all extended modes.

Informational options

This class of command line options can be called without <MODE> or <INFILE> parameters with the following syntax:

```
> IeeeCC754++ <OPTION>
```

The available options can be used to retrieve information about basic usage, the supported modes, or the current version of IeeeCC754++. The following options are supported:

- --help: Shows general usage information.
- --modes: Displays and explains the different testing modes supported by IeeeCC754++. This corresponds to the <MODE> parameter.
- --args or --options: When one of these options is used, a list of the command line options supported by IeeeCC754++ is shown. This includes all options discussed in this section.
- --version: This option causes IeeeCC754++ to show information on the current build of IeeeCC754++ itself. The output is identical to that produced by IeeeCC754++ -t (cf. Listing 3.15).

FTZ

When IeeeCC754++ is executed in a floating-point environment not supporting subnormal numbers, it will report errors for all test vectors whose results are in the subnormal range. While reporting these errors is certainly the correct behaviour, it does not yield extra insight since it was known beforehand that subnormal values are not supported (and the floating-point environment is not fully IEEE-conforming).

When subnormal numbers are not available in a floating-point environment, a result that would be subnormal after rounding must be set to a representable floating-point number, i. e. it must be set to zero.[4] IeeeCC754++ offers the possibility to check whether rounding before obtaining a subnormal number is performed correctly, i. e. in an IEEE-conforming manner, and whether values are consistently flushed to zero. Within IeeeCC754++, this behaviour is achieved by altering the correct result of the corresponding test vector to a zero with unchanged sign and then proceeding with testing as with unmodified test vectors. The option --ftz enables this behaviour, while --noftz explicitly restores the default behaviour.

Since there is no general definition how flushing to zero is to be performed (such a definition is certainly not part of IEEE 754-2008 as the standard aims at providing

[4] FTZ is usually not influenced by the currently chosen rounding mode: First, the result is computed and rounded. If the result is subnormal after rounding (i. e. it is smaller in magnitude than the smallest normalised number, but larger than the respective zero), it then is set to zero, see e. g. [INT17b] or [Cas08].

the most accurate results possible, i. e. it requires conforming implementations to support subnormal numbers), there is ambiguity in the flushing process due to floating-point zeroes being signed. Therefore, implementers can either flush every subnormal result to +0 regardless of the number's sign, or they can respect the sign and flush to −0 in case of a negative subnormal result. `IeeeCC754++` supports both variants with the command line options `--ftz` which flushes to +0 and `--ftzsigned` which correctly flushes to signed zeroes.

Some floating-point implementations avoid handling subnormals in an even stricter way by not only flushing subnormal results to zero, but also replacing subnormal input values with zeroes (also called DAZ or "denormals are zero", cf. e. g. [Cas08]). `IeeeCC754++` does not support DAZ due to its testing model via test vectors: Setting input operands to zero changes the value of the (correct) result and might also influence (and therefore change) which exceptions should be returned. Since `IeeeCC754++` has no knowledge of the relation between input values and correct result apart from that encoded in the test vectors, it is not possible to adapt the expected result and exceptions according to the changed input operand(s).

However, `IeeeCC754++` provides the command line option `--skipsubnormal` (and its synonymous option `--skiptiny`) which causes the testing engine to skip all test vectors containing subnormal values, either in the operands or the result.

ULP thresholds

As discussed in Section 3.1.12, `IeeeCC754++` tries to determine if an incorrect returned result is only rounded incorrectly, i. e. if it is only a few ulps off, or if a more serious error occurred. The default threshold for regarding the difference between the correct and the returned result as critically wrong is 8 ulps (or, in other words, the last three binary digits being wrong).

However, when the floating-point result is computed correctly (to infinite precision) and only rounded incorrectly afterwards (to target precision), the difference between correct and returned result will only be 1 ulp. In order to allow for analysis of these tighter requirements, the threshold can be changed via the command line parameter `--ulp=<ULP>` where `<ULP>` is the new threshold in ulps.

Message digests

`IeeeCC754++` offers the option to calculate hash digests of the log files generated during the testing process in verbose mode. For a chosen digest mode `<DIGEST>`, an additional output file called `<LOGFILE>.<DIGEST>` will be written containing the hashed messaged digest of `<LOGFILE>`.

The syntax for these options is as follows:

```
> IeeeCC754++ ... --digest[=<DIGEST>]
> IeeeCC754++ ... --hexdigest[=<DIGEST>]
```

with <DIGEST> being the chosen digest algorithm. If <DIGEST> is not given, the default digest SHA1 will be used.

Both variants write the message digest into the output file; however, they generate different formats: `--digest` writes the digest in binary form, whereas `--hexdigest` generates a hexadecimal representation of the same value.

Message digests via the described options are only available in verbose mode. It should be noted that while conceptually similar to fingerprint mode, the generated fingerprint is considerably different: A *fingerprint* produced in verbose mode with one of the digest options contains no summary information and may be written in hexadecimal representation, whereas the digest part of a fingerprint generated in fingerprint mode always contains a header with additional information and is written in binary format. Furthermore, the source content for which the digest is calculated differs: In verbose mode, it consists of the output file in one of the verbose output formats described in Section 3.3.2, while in fingerprint mode, only error information in checksum mode format is used to compute the digest.

Suppressing log file generation

Via the option `--nolog`, it is possible to suppress the generation of a log file. While the option is available in all extended modes, it is mainly useful in verbose mode when used together with one of the message digest options `--digest` or `--hexdigest`. In this case, it is possible to avoid writing the (considerably larger) log file to disk while still being able to generate a file containing the hash digest.

When using `--nolog` without generating a hash digest, `IeeeCC754++` behaves exactly as in quiet mode, see Section 3.3.5.

Mode and option overview

We conclude the discussion of `IeeeCC754++`'s testing modes and command line options by referring to Appendix C.2 where a short overview of the modes and options is given.

3.4 The evaluation framework

The goal of `IeeeCC754++` and its included set of tools is to help an end user to evaluate the IEEE-conformity of a given floating-point environment and therefore the expected numerical quality that can be achieved in this environment (or to identify potential problems). While it is possible to execute single test runs, in many cases it can be desirable to test different combinations of FPUs, compilers, and floating-point formats in order to assess which combination should be given preference (with the usual tradeoff between performance and numerical quality in mind). It can even be sensible to test the current environment with different

architectures such as the default architecture in addition to the native architecture of that specific environment.

This section starts with a description of the framework's structure and its configuration files which control the testing and evaluation process. Afterwards, we describe the analysis modules built into the evaluation framework and the process of adding (and calling) custom modules that can be used for a deeper analysis when necessary. We conclude this section by presenting several small helper scripts that further ease the setup and execution of a large number of test runs, including the graphical tool `IeeeCC754++LogViewer` which provides a convenient way of viewing the (possibly large amount of) output generated by the framework in as much detail as required.

3.4.1 Using the evaluation framework

Every test run (including the analysis of the results) consists of a number of actions that need to be performed, such as compiling a suitable `IeeeCC754++` instance and executing test runs with appropriate testsets. The evaluation framework abstracts these actions into separate tasks:

- Job task: A *job* consists of a list of tasks to be executed. All tasks explained in this section (and shown in Table 3.3) can be called as jobs, including a job task itself which will then be executed as a sub job.
- Compile task: A *compile task* configures and builds `IeeeCC754++` for the specified set of parameters, including the target architecture, a compiler with compiler options, and FPUs.
- Execute/test task: A *test task* executes an `IeeeCC754++` binary for the specified FPUs and test vector sets. Since an appropriately built executable is mandatory, a compile task file is needed as a parameter, and the corresponding compile task must have been executed prior to calling the test task.
- Evaluation task: An *eval task* analyses the results of a previously (successfully) executed test task. The parameters needed for an eval task consist of a test task file and an evaluation function that is applied to the output files generated by the test task. All information needed to locate the log files is retrieved from the test task file.

Additionally, the evaluation framework knows two "fake" tasks which can be used for technical reasons:

- Mode task: The evaluation framework allows for the application of three of the testing modes described in Section 3.3: the classic mode (`-u`, see Section 3.3.1) and the verbose mode with and without consideration of

exceptions (`-v` and `-vix`, cf. Section 3.3.2). The default mode is "full" verbose mode (`-v`). To change the mode used during the testing process, a *mode task* with an appropriate parameter can be executed.

- Delimiter task: In addition to the output files generated by the different tasks, a master log is generated containing information about the executed tasks and the results of the evaluation functions called by eval tasks. In order to structure this log, a delimiter can be added to this output with a *delimiter task*.

For completeness reasons, we briefly mention a supplementary task belonging rather to the optimisation framework (cf. Section 3.5), but nonetheless connected to the evaluation framework:

- Optimisation task: The optimisation framework is a variation of the evaluation framework specifically tailored to studying the influence of different compiler options on runtime and floating-point conformity. Since the evaluation framework provides the technical basis for the optimisation framework, *opt tasks* can be used as input to the evaluation framework. An extensive description of the optimisation framework can be found in Section 3.5, together with a detailed specification of opt jobs.

Table 3.3 shows the tasks supported by the evaluation framework, together with their descriptors which are needed inside job task files and the file formats that define the possible parameters for the different tasks.

Descriptor	Task name	Purpose	File format
`j`	job	Call other tasks.	`<FILE>.job`
`c`	compile	Build an executable.	`<FILE>.com`
`t`	test	Execute a test.	`<FILE>.test`
`e`	eval	Evaluate results of a test.	`<FILE>.eval`
`m`	mode	Change testing mode.	–
`d`	delimiter	Add delimiter into output.	–
`o`	opt	Call the optimisation framework.	`<FILE>.opt`

Table 3.3: *Tasks supported inside the evaluation framework.*

The order of tasks executed during an evaluation framework run are controlled by the contents of a "master" job file. To start a run, the `Python` script `job.py` must be called with such a master job file which includes a list of tasks (and corresponding configuration files). This list is then executed in the order given in the job file. Listing 3.16 shows the syntax of a `job.py` invocation including all possible options.

```
> python job.py --help
Looking for SQLite and pySQLite.
Try system install (sqlite3)... found SQLite v3.7.17 and pySQLite v2.6.0.
job.py: IeeeCC754++ job starter

Usage is:
  job.py [-m <MODE>] [-e <EVALFUNC>] [-d <DEBUGLEVEL>] [-i] [-q] [-s] <JOBFILE>
  job.py -l|--list
  job.py -h|--help

where
  -d <DEBUGLEVEL>   sets the level of debugging output:
                     -1 equivalent to -q [quiet]
                      0 normal
                      1 show "file found" messages
                      2 show path information
                      3 show info about tasks/actions etc.
                      4 full output: e.g. show content of generated scripts
  -m <MODE>         sets the output mode:
                      c   classic IeeeCC754 mode (no eval!)
                      v   verbose mode (default)
                      x   verbose mode, ignore exceptions
  -e <EVALFUNC>     set evaluation function (overriding settings in files)
  -i                show information about IeeeCC754++ executables
  -q                quiet: suppress almost all job.py output
  -s                show output of subprocesses (compile & test)
  -l                list all known eval functions
  <JOBFILE>         name of the job file whose contents will be executed
```

Listing 3.16: *Output of `job.py -help`*

The only mandatory parameter is the master `<JOBFILE>` that controls which further tasks are called and in which order. Additionally, there are three main groups of parameters:

- Parameters influencing the testing and evaluation process: `-m <MODE>` changes the default `IeeeCC754++` testing mode to `<MODE>`, whereas the evaluation function used during the analysis stage (which is usually set in the eval files that are called by the master job) can be overridden with `-e <EVALFUNC>`.

  Note that when choosing a specific evaluation function via the `-e` parameter, all eval tasks will only use this selected function to analyse the log files. It offers an easy way to temporarily change the evaluation function used during a framework run. On the other hand, `-m` only changes the *default* testing mode, i. e. when a mode task is encountered in a job file, the testing mode is then changed to the mode specified in the mode task.

- Parameters influencing the output of the evaluation framework, especially what is printed and with how much detail: `-q` selects quiet mode, i. e. the amount of output is reduced compared to a normal run.

  The output of the subprocesses that actually perform the build and test tasks is only shown when these subprocesses return a non-zero exit value,

indicating that something went wrong during the execution. The option `-s` causes the evaluation framework to display this output unconditionally, thereby enabling the possibility of tracing exactly what happens during the runtime of each subprocess.

When `-s` is not specified, the parameter `-i` can be used to execute a small helper script that runs the built `IeeeCC754++` binary in order to retrieve information about this executable.

Finally, when the evaluation framework does not behave as expected and some debugging information is needed in order to analyse the faulty behaviour, the amount of (internal) information printed can be increased by specifying suitable debug levels via `-d <DEBUGLEVEL>`. Possible values and their explanation are listed in Listing 3.16.

- Finally, there are two parameters serving informative purposes which are used without a `<JOBFILE>`: `-h` or `--help` prints the usage information shown in Listing 3.16, and `-l` or `--list` displays a list of all evaluation functions available to the evaluation framework, including custom functions. For an example of such output and for details on implementing custom evaluation functions, see Listing 3.32 and Section 3.4.2.

Prerequisites

In order to implement a light-weight framework that is easily extensible, the evaluation framework has been implemented in `Python`. Since `Python2` is the most widespread version and `Python3` is not available on some older platforms, `Python2` was chosen as the framework's language. Therefore, a suitable `Python2` interpreter must be installed on the platform on which the evaluation framework is executed. The minimum required `Python` version for the evaluation framework is 2.5. Furthermore, the following modules (which should be built into every `Python2` installation) are necessary:

- Modules for basic OS support and properties: `os`, `os.path`, `platform`, `sys`.
- Modules to call external functions or commands: `subprocess`, `ctypes` (`Python` version 2.5 and newer), `dl` (`Python` versions before 2.5).
- Modules for efficient use of `Python`'s internal data types: `copy`, `time`.
- To enable an efficient way of checking which of the log files generated by `IeeeCC754++` are identical, the evaluation framework calculates checksums of these log files via message digests. Therefore, the module `hashlib` is needed.
- In order to use `IeeeCC754++LogViewer` (see Section 3.4.3), `wxPython` version 2.8 must be installed.

The evaluation framework executes the compile and test tasks via shell scripts. These assume that GNU `bash` version 2 or newer is installed, together with the command line tools `ldd` and `make`. Obviously, a compiler with corresponding linker is needed as well.[5]

The prerequisites discussed so far should be relatively easy to meet. There is however one requirement which might not be pre-installed by default: The analysis process inside the evaluation framework uses an in-memory database for efficient storage, retrieval, and evaluation of the results. Since it is in widespread use[6], `SQLite` [SQL16a] was chosen as the database system. Consequently, `SQLite` must be installed on the platform, as well as a corresponding `Python` module (the modules `sqlite3` and `pysqlite2` are suitable) to provide a native interface to the evaluation framework.

Unfortunately, not all platforms provide such an `SQLite Python` module. In order to enable easy installation (and execution) of the evaluation framework, a custom module called `importPySQLite.py` is provided that searches the current system for a suitable `Python` module, imports it into `Python` when it is found, and tries to build it from source (which is also contained in the `IeeeCC754++` source tree, see Figure 3.2) otherwise. Although this module is automatically imported when running the evaluation framework, it can also be used as a standalone script to enable advance testing of whether the current platform supports a suitable `SQLite` module. Listing 3.17 shows output of a successful standalone execution, i. e. on a platform where `SQLite` including a corresponding `Python` module is installed and works as expected.

```
> python importPySQLite.py
importPySQLite.py: Trying to import SQLite module.
Looking for SQLite and pySQLite.
Try system install (sqlite3)... found SQLite v3.7.17 and pySQLite v2.6.0.
Running test procedure for importSQLite.py:
(u'arm', u'neon')
(u'bgq', u'qpx')
(u'x86', u'avx')
(u'x86', u'x87')
Deleted 4 rows.
```

Listing 3.17: *Output of successfully importing `SQLite`.*

Finally, it is important to understand how the evaluation framework uses and locates the testsets for the test runs: Since the evaluation framework employs `IeeeCC754++`'s verbose mode to generate easily parsable log files as a basis for the analysis stage, the testset files that are fed into `IeeeCC754++` must be in UCB format. Furthermore, the location of these files cannot be arbitrarily chosen: They

[5] Actually, the helper scripts described in Section 3.4.2 need some more command line tools such as `cp`, `grep`, `date`, `mkdir`, or `rm`. However, these should be available on virtually any platform that `bash` has been installed on.

[6] `SQLite` claims to be the most widely deployed and used database engine, see [SQL16b].

must be located in the directory `testsets/` inside the `IeeeCC754++` source tree. Appropriate testset files in UCB format can be generated from the testset files in Coonen syntax that are provided by `IeeeCC754++` in the directory `src/testsets/`. For details on how to convert testsets in manual or automated manner and prepare corresponding testset files for use in the evaluation framework, see Section 4.7. For convenient testing of all operations with the most common floating-point formats, `IeeeCC754++` includes the files `allh`, `alls`, `alld`, `alll`, and `allq` in `testsets/`, enabling testing of the half, single, double, extended, and quadruple formats (cf. Table 1.1) by specifying the names of the testset files in the corresponding test and eval task files, see next section.

Configuration files

For a typical (single) run of the evaluation framework, a set of four configuration files is needed: a job task file controlling the execution of the sub tasks, a compile task file building `IeeeCC754++`, a test task file running the newly created executable with the specified testset(s), and an evaluation task file applying at least one evaluation function to the log files generated by the test task. In this section, we discuss the structure of these configuration files and the possible parameters. Heavily commented example files for the different tasks can be found in Appendix C.5.

The basic syntax rules for the configuration files are defined as follows:

- All lines beginning with "`#`" are comments.
- Comments and empty lines (i. e. lines only containing whitespace) are ignored.
- Parameters must be contained on a single line and must be given in the form "`<KEY> = <VAL>`".
- `<KEY>` denotes the name of the parameter and `<VAL>` the given value.
- `<KEY>` includes everything before the first "`=`"; everything after that character is considered as `<VAL>`.
- `<KEY>` and `<VAL>` will be trimmed before evaluation, i. e. whitespace at the beginning and end of these values is removed.
- Unknown parameters (i. e. unknown values for `<KEY>`) will be ignored, but a warning will be printed.
- The case of `<KEY>` is irrelevant.
- The case of `<VAL>` is taken "as is", i. e. no changes will be applied.

- Lines that are neither a comment (or empty) nor contain a valid <KEY>/<VAL> pair are ignored, but a warning will be printed.
- File names (such as the name of environment scripts, see below) must not include spaces.

Job task files A job file consists of a list of tasks that will be executed in the given order. A list of valid tasks can be found in Table 3.3. Note that job files are the only type of configuration files which follow a different, simpler syntax:

- Lines must have the form "<CHAR> <VAL>" where <CHAR> is a single character, followed by whitespace, and <VAL> the rest of the current line, excluding trailing whitespace.
- Only the job descriptors listed in Table 3.3 are allowed.
- For job, compile, test, and eval tasks, <VAL> is the name of the corresponding configuration file. The appropriate file ending is automatically appended to the name if not specified.
- For a mode task (<CHAR> is equal to m), only the values v, x, or c are allowed, switching the testing mode to either verbose mode (v), verbose mode without considering exceptions as errors (x), or classic mode (c).
- When a delimiter job is called, a section delimiter is printed into the log file, including the given string <VAL> as heading.

Since job tasks are valid parameters inside a job file, it is possible to write a master job file containing a list of further job files as sub tasks. If several jobs are set up in this manner, one can easily enable or disable certain test runs while calling the evaluation framework with the same job file by simply (un)commenting the appropriate sub tasks in the master job file.

Listing 3.18 shows an example of a simple job file that first prints a delimiter with the string "EXAMPLE - EVALUATION FRAMEWORK" into the log file, and then compiles IeeeCC754++ with the settings given in the compile file ex.com. Before executing tests or analysing results, the testing mode is changed to "verbose". Afterwards, IeeeCC754++ is executed according to the parameters in ex.test, before finally the resulting log files are analysed and evaluated as configured in ex.eval.

Note that for this example to work, the compile task ex.com must be a parameter in ex.test and that ex.test must be a parameter inside ex.eval (see below).

```
# show delimiter including the heading "EXAMPLE - EVALUATION FRAMEWORK"
d EXAMPLE - EVALUATION FRAMEWORK
```

```
# compile IeeeCC754++
c ex

# change IeeeCC754++ testing mode to verbose
m v

# execute test and eval tasks
t ex
e ex
```

Listing 3.18: *Example job task file.*

Compile task files In order to build IeeeCC754++, a number of settings are needed, in particular the architecture and FPUs to build for, the compiler that should be used, and relevant compiler and environment settings. All valid parameters are listed in Table 3.4, together with default values.

Parameter	Mandatory?	Default	Purpose
ENV	no	—	Environment scripts.
MODULE	yes	—	Environment modules.
ARCH	yes	—	Architecture.
FPU	no	—	List of FPUs.
CFLAGS	no	—	Flags for the C-compiler.
CPPFLAGS	no	—	Flags for the preprocessor.
CXXFLAGS	no	—	Flags for the C++-compiler.
LDFLAGS	no	—	Flags for the linker.
LIBS	no	—	Extra libraries.
ARGS	no	—	Extra arguments passed to configure.
CORES	no	1	Number of parallel build processes.
MODE	no	main	Build mode.
BITS	no	—	32 or 64 bit build?

Table 3.4: *Compile task file parameters.*

For the meaning of MODE and BITS, see Appendix A.3. If extra options are needed for the configuration process, they can be passed to configure by adding them to ARGS.

The mechanism that is used to choose a compiler (and its relevant compiler options) requires some explanation: As described in Appendix A.3.4, page 297, the compiler can be chosen by setting the environment variables MYCC and MYCXX (or alternatively CC and CXX). While this can be done by setting the values in the global environment before executing an evaluation framework run, this approach does not lend itself to batch testing of different compiler actions without user interaction. Therefore, IeeeCC754++ supports so-called *environment scripts* which are bash scripts that are sourced into the bash scripts executed by the evaluation

framework (or, more precisely, the corresponding compile and test jobs). Inside an environment script, the chosen values for the compiler can be set and exported, see Listing 3.19.

```
#!/bin/bash
export MYCC = gcc-4.7
export MYCXX = g++-4.7
```

Listing 3.19: *Example environment script* `gcc47_env.sh`.

This approach is necessary to set the compiler in an automated way, but introduces an alternative way of setting the environment variables that influence the compilation and linking stages during the build process, namely `CFLAGS`, `CPPFLAGS`, `CXXFLAGS`, `LDFLAGS`, and `LIBS`. These can either be exported in the environment script or set as parameters of the compile file.

Note that the environment script that is specified in `ENV` can also be used for further setup that might be necessary to enable execution in the current testing floating-point environment, such as setting up accelerators or adapting the search path for shared libraries. Furthermore, more than one environment script may be given via `ENV`. Each of the given scripts is then sourced in the given order prior to configuring and building `IeeeCC754++`. For every environment script that should be sourced, a separate line with an `ENV` entry must be specified.

Additionally, the evaluation framework supports so-called Environment Modules (see [MOD17; WIK17k]) which manipulate the current environment as needed for the specified module. Every module that should be loaded must be specified as a separate line with a corresponding `MODULE` entry.

Listing 3.20 shows an example of a compile script using the environment script `gcc47_env.sh` (cf. Listing 3.19) that builds `IeeeCC754++` with `gcc` 4.7 for the default architecture with the additional `C99` FPU (see Section 5.1.1), using four parallel build processes during compilation.

```
ENV = gcc47_env.sh
ARCH = default
FPU = c99
CORES = 4
```

Listing 3.20: *Example compile task file.*

Test task files For every compile task, there should be (at least) one test task that runs the `IeeeCC754++` executable with the specified testsets for every FPU (usually including the main FPU) that `IeeeCC754++` was built for. Table 3.5 shows the parameters known to a test task.

Since a test task needs an executable built by a compile job as well as some of the settings used during the build process (such as the architecture that `IeeeCC754++` was built for), a `COMPILE` parameter is mandatory that contains the

Parameter	Mandatory?	Default	Purpose
ENV	no	—	Environment scripts.
MODULE	yes	—	Environment modules.
COMPILE	yes	—	Compile task.
TESTSET	yes	—	List of testsets.
FPU	no	main	List of FPUs.
ARGS	no	—	Extra arguments passed to IeeeCC754++.
BATCH	no	—	Batch system for remote execution.
EXECPREFIX	no	—	Execution prefix, e. g. mpiexec call.

Table 3.5: *Test task file parameters.*

corresponding compile task file. The test job then extracts those values that are needed for the test run.

While it would be possible to extract the values of the environment scripts used during compilation from this compile task file, it might not always be desirable to use these same scripts during IeeeCC754++ execution. To allow for more flexibility when configuring an evaluation framework run, environment scripts that should be sourced before executing the actual test run must be explicitly specified via ENV. That being said, in most cases the scripts passed here will be identical to those given in the corresponding compile file. Like in the compile task case, each environment script must be specified on a separate line by adding an ENV value. The same reasoning applies to Environment Modules, and as a consequence, all modules needed to execute the previously built IeeeCC754++ must be specified via separate MODULE lines.

If additional parameters should be passed to IeeeCC754++ as command line options (e. g. to choose between accelerator devices), these can be specified in ARGS.

TESTSET and FPU take lists of testsets and FPUs, respectively. IeeeCC754++ will then be run with all combinations of testsets and FPUs.

Not all user environments allow for direct execution of executables, but require the use of a batch system, such as most supercomputers, or can only be executed through a wrapper executable such as mpiexec for execution on several hosts via MPI. The latter case is addressed by the parameter EXECPREFIX that denotes the string that the execution command line should be prefixed with, i. e. for a value of EXECPREFIX = mpiexec -n 2 --hostname host1,host2 to execute IeeeCC754++ via MPI on the hosts host1 and host2, the command line "mpiexec -n 2 --hostname host1,host2 IeeeCC754++ <PARAMETERS>" will be executed instead of "IeeeCC754++ <PARAMETERS>".

For batch system execution style support, the evaluation framework includes modules for some of the most widely spread batch execution systems, namely LoadLeveler [IBM01], PBS/Torque [PBS16; Sta06], and SLURM [YJG03]. When

the corresponding values `ll`, `pbs`, or `slurm` are given in `BATCH`, the evaluation framework generates adequate job files, submits the jobs into the execution queue, waits for the end of the job, and finally retrieves the result. Therefore, it might be advisable to limit the number of test tasks executed on such an environment, depending on the waiting time before the batch job is scheduled to run in the environment.

Listing 3.21 shows an example test file that executes the `IeeeCC754++` binary produced by the compile job `ex.com`. For consistency reasons, the same environment file is sourced. Both FPUs available in the `IeeeCC754++` executable (`main` and `C99`) are tested with the testsets `t1s` and `t1d` (see Listing 4.6). However, the test script generated by the test task (which calls `IeeeCC754++` for the four combinations of testset and FPU) is not executed directly in the environment that the evaluation framework runs on, but through Slurm.

```
ENV = gcc47_env.sh
COMPILE = ex.com
TESTSET = t1s t1d
FPU = main c99
BATCH = slurm
```

Listing 3.21: *Example test task file.*

Eval task files After completing compile and test tasks, the resulting log files can be fed into the evaluation framework for analysis and evaluation. In contrast to compile and test tasks, an eval task only requires a minimal amount of information, namely the log files and at least one evaluation function that will be applied to these files. For maximum flexibility, we provide the following three ways of executing an eval task:

- Standalone execution: The eval task part of the evaluation framework can be called as a standalone script which analyses one log file with the specified evaluation functions. For details, see below.

- Specifying a test task: When a test task is given as a parameter in the eval task file (`TEST`, see below), the evaluation framework retrieves the log files generated during the runtime of the test task and applies the requested evaluation functions to these files.

- Alternatively, a list of log files (either with relative or absolute path) and an optional log path can be given together with the evaluation functions. The evaluation framework then tries to open the files specified in `LOGFILES` and `LOGPATH` and analyses them.

When both an eval task and a list of log files are specified in the eval task file, the eval task file parameter takes precedence, i. e. the evaluation framework

Parameter	Mandatory?	Default	Purpose
EVALFUNCTION	no	basic	List of evaluation functions.
TEST	no	—	Test task.
LOGFILES	no	—	List of log files.
LOGPATH	no	—	Path to log files.

Table 3.6: *Eval task file parameters.*

ignores the parameters LOGFILE and LOGPATH in this case. Table 3.6 shows the (short) list of parameters allowed in eval task files.

Each log file can be analysed with more than one evaluation function by either specifying multiple evaluation functions, using multiple eval task files, or by using aggregated evaluation functions, see Section 3.4.2.

```
EVALFUNCTION = basic
TEST = ex.test
```

Listing 3.22: *Example eval task file.*

Executing tests

When an evaluation framework run is performed, i. e. when job.py is executed with an appropriate job task file, job.py writes a number of status messages and the output of the evaluation function reports to the console. It is advisable to redirect this output into an appropriate log file for further reference. By default, the evaluation framework informs the user about the currently performed action, the log files and evaluation reports that were generated (and written into files), and the output of these reports. However, with the command line switches shown in Listing 3.16, the amount of output can both be increased (via the options -s, -i, and -d) or decreased (-q).

During an evaluation framework run, the following information is recorded for later analysis:

- Errors encountered, e. g. when compile tasks fail or an IeeeCC754++ executable cannot be executed.
- Summaries of the analysed log files.
- Names of the generated files, especially the output files of the evaluation function reports.

After all tasks in the master job file have been executed, the evaluation framework generates and prints a number of summaries in order to enable a quick overview of testing results:

- A global summary, including the name of the corresponding log file, a hash message digest of this file, and the summarised test results.
- The same summary, but sorted according to the name of the test file.
- A list of all evaluation report files, grouped by the evaluation function that was used to create the analysis result.
- A list of all errors that were encountered during runtime. This list provides a very quick means to verify whether all tasks were properly executed (when the list is empty) or to get pointers at potential problems.
- Finally, the testing summary that was already printed at the top of the summary list is displayed again, this time sorted according to the hash value of the corresponding log file. This approach is especially helpful when different compiler versions are tested for the same architecture as identical log files can be quickly identified because of the sorting order.

A detailed example including these summaries can be found in Section 6.1.

When the tasks that implement the evaluation framework functionality are executed, a large number of files is created, such as the directory tree for the configuration and compilation of a suitable `IeeeCC754++` binary or the log files generated by a test task. In order not to clutter the directory containing the evaluation framework (cf. Section 3.4.1), all generated files are collected in the subdirectory `run/`. Whenever a run of the evaluation framework is performed and a job task is encountered, a corresponding subdirectory (called `run_<JOBNAME>_<DATE>`) is created directly in `run/` (for the master job file) or in the subdirectory in which the current job resides. For compile, test, and eval tasks, additional subdirectories of the currently active job task are created. All log files and evaluation function reports (as well as the executables and scripts used to generate these files) can therefore be found in this directory. Since the paths involved tend to get long and cryptic, the full names of the generated files are written to the console for easy access (see above).

Standalone evaluation

As discussed in the last section, an eval task requires only a log file and an evaluation function as parameters. Therefore, the script `eval.py` can be called as a standalone script with the parameters shown in Listing 3.23. Syntax and parameters are mostly identical to those of `job.py` (cf. Listing 3.16), with one notable difference: `<FILE>` can either be an eval task file (named `<FILE>.eval`) or a log file. In the latter case, the file is directly passed to the evaluation stage, whereas in the first case, the eval task file is parsed as usual. In particular, the log files to be analysed must be specified via the `LOGFILES` parameter. The method

of automatic retrieval of log files via a test task file (specified with the parameter TEST) is not permitted in standalone mode.

```
> python eval.py --help
Looking for SQLite and pySQLite.
Try system install (sqlite3)... found SQLite v3.7.17 and pySQLite v2.6.0.
eval.py: IeeeCC754++ job analyser

Usage is:
  eval.py [-e <EVALFUNC>] [-d <DEBUGLEVEL>] [-q] <FILE>
  eval.py -l|--list
  eval.py -h|--help

where
  -d <DEBUGLEVEL>   sets the level of debugging output:
                     -1 equivalent to -q [quiet]
                      0 normal
                      1 show "file found" messages
                      2 show path information
                      3 show info about tasks/actions etc.
                      4 full output: e.g. show content of generated scripts
  -e <EVALFUNC>     set evaluation function (overriding settings in files)
  -q                quiet: suppress almost all eval.py output
  -l                list all known evaluation functions
  -h                this message
  <FILE>            name of either
                    * an .eval file whose contents will be executed
                    * a log file that will directly be analysed
```

Listing 3.23: *Output of `eval.py -help`*

Note that when passing a log file directly to `eval.py`, the only way of choosing an evaluation function consists of specifying the desired one with the command line option `-e <EVALFUNC>`. A list of all available evaluation functions as retrieved with `python eval.py -list` can be found in Listing 3.24.

```
> python eval.py --list
Looking for SQLite and pySQLite.
Try system install (sqlite3)... found SQLite v3.7.17 and pySQLite v2.6.0.
eval.py: IeeeCC754++ job analyser

Known eval functions:

all  =>  all v0.1
basic  =>  basic v0.2
basic_near  =>  basic_near v0.1
detailed  =>  detailed v0.2
error_list  =>  error_list v0.2
error_report  =>  error_report v0.2
example  =>  Example evaluation function v0.01
operation_report  =>  operation_report v0.2
opteval  =>  Evaluation function for the optimisation framework v0.01
roundings  =>  roundings v0.2
ulp  =>  ulp v0.1
very_detailed  =>  very_detailed v0.2
```

Listing 3.24: *Output of `eval.py -list`*

Code structure

For reference purposes, we briefly discuss the code structure of the evaluation framework. It is located in the `IeeeCC754++` source tree in the directory `eval/`; the files and directories implementing the evaluation framework are shown in Figure 3.2.

```
eval/
├── evalfunc/
├── files/
├── run/
├── submit/
│   ├── ll.py
│   ├── pbs.py
│   └── slurm.py
├── action.py
├── job.py
├── compile.py
├── test.py
├── eval.py
├── globals.py
├── settings.py
├── structures.py
├── tools.py
├── importPySQLite.py
├── pysqlite_src/
│   ├── buildPySQLite.sh
│   ├── pysqlite-2.6.3.tar.gz
│   └── sqlite-autoconf-3071300.tar.gz
├── genJobs.py
├── startTests.sh
├── checkErrors.sh
└── updateJobs.sh
```

Figure 3.2: *Code structure of the evaluation framework.*

The source code consists of files implementing the different tasks (and corresponding base classes in `action.py`), the batch modules needed for remote execution of test tasks (all files in `submit/`), and global settings and structures. Furthermore, `importPySQLite.py` and the corresponding directory `pysqlite_src/` implement the import functionality for `SQLite` described in Section 3.4.1.

The different evaluation functions usable for the analysis of the log files produced by `IeeeCC754++` are implemented in the directory `evalfunc/`; for details on these functions and how to implement custom evaluation routines, see Section 3.4.2. The directory `files/` contains reference examples of task files, see Appendix C.5.

The directory `run/` is created whenever an evaluation framework run is performed and it does not already exist. During the runtime of the framework, `run/` is filled with all files generated by the corresponding job, compile, test, and eval tasks.

Finally, the four files shown at the end of Figure 3.2 implement useful helper scripts which are described below, see Section 3.4.3.

3.4.2 Analysis modules

The evaluation framework contains extensive analysis facilities in order to enable easy analysis and evaluation of a large number of test runs. Each test execution consists of an `IeeeCC754++` run in verbose mode that generates a log file containing information about every test vector encountered during the testing process. The evaluation framework parses these log files and pushes the retrieved information into an in-memory `SQLite` database. This approach provides a fast and simple way to extract and aggregate the floating-point errors and properties of the test run via a standardised interface (i. e. via SQL queries).

The evaluation framework uses *analysis modules* to analyse the floating-point properties of the underlying floating-point environment with the data stored in the in-memory database. Inside these modules, an *evaluation function* retrieves and aggregates the relevant information via an SQL call. Additionally, further processing is applied for extensive evaluation and to generate a human readable report of the analysis results. The evaluation framework provides a number of evaluation functions that cover a range of different relevant properties.

The information generated by an evaluation function is dropped into a log file for later access. Furthermore, the SQL table containing the parsed contents of the test vector file is saved in CSV format[7] to a corresponding `.csv` file in case further (external) analysis is required.

Each analysis module contains exactly one evaluation function tailored for a specific analysis task. Additionally, it is possible to aggregate the output of several evaluation functions from different analysis modules for a more extensive inspection of a single test run (see e. g. the `detailed` evaluation function below).

In this section, we describe the analysis modules contained in the evaluation framework. When the supplied evaluation functions are not sufficient for the desired evaluation of a floating-point environment, it is possible to extend the evaluation framework with custom evaluation functions. This process is also described later in this section.

An overview of the pre-installed evaluation functions can be retrieved with the `--list` parameter of `job.py` or `eval.py`, see Listings 3.16 and 3.23; example output can be found in Listing 3.24. Figure 3.3 shows the files that implement the different evaluation functions as `Python` modules: `__init__.py` and

[7] Comma Separated Values: a plain text table format that separates values by a delimiter (originally a comma ","). `IeeeCC754++` uses a semicolon ";" as delimiter.

`list_evalfuncs.py` initialise the `evalfunc` module and implement the listing functionality. `evaltools.py` and `error.py` provide common functionality and information about the different error types that can occur during an `IeeeCC754++` test run; the provided functions can be used in the analysis modules. Finally, all other files are named after an evaluation function and contain the corresponding implementation, e. g. the file `basic.py` contains the `basic` evaluation function.

```
eval/
└── evalfunc/
    ├── __init__.py
    ├── all.py
    ├── basic.py
    ├── basic_near.py
    ├── detailed.py
    ├── error_list.py
    ├── error_report.py
    ├── errors.py
    ├── evaltools.py
    ├── example.py
    ├── list_evalfuncs.py
    ├── operation_report.py
    ├── opteval.py
    ├── roundings.py
    ├── ulp.py
    └── very_detailed.py
```

Figure 3.3: *Code structure for the analysis modules inside the evaluation framework.*

Available evaluation functions

While every analysis module presents itself as a single module to the evaluation framework, there are actually two types of modules: the modules that retrieve and analyse the data stored in the database, and the modules that aggregate the reports of other modules into a single report. We discuss the "analysing" modules in detail and then briefly describe the aggregation modules.

In the following, *success* describes the execution of the operation contained in a test vector that returned the correct floating-point result and raised all expected exceptions (and no other exceptions). Consequently, an *error* describes an execution where the returned result is not correct or wrong exceptions were returned. Note that in `-vix` testing mode, *error* only refers to cases where the returned result is not correct, regardless of what exceptions are (or are not) returned (cf. Section 3.3.2). Although the information about exceptions is still contained in the log file, all test vectors which returned the correct floating-point

number are counted as a success (even when errors concerning exceptions occurred). All examples shown in this section were generated with `IeeeCC754++` built for the default architecture and executed on an x86 platform. The testset contained only test vectors for the addition and `fma` operations.

`basic` The `basic` evaluation function provides basic information about the underlying `IeeeCC754++` run and its testing results. It calculates the success rate for each operation contained in the testset, as well as an overall success rate. Furthermore, the success and error counts are shown.

In addition to the operation success rates, the `basic` module aggregates the operations (and their respective counts) into groups: *basic* (+, −, ∗, /), *extra* (remainder, square root, and `fma`), *conv* (all conversions involving floating-point and integer formats), *output* (conversions between binary and decimal formats), and *elem* (elementary functions such as trigonometric, power, root, exponential, and logarithmic functions), cf. Section 4.1.2.

Example output can be found in Listing 3.25. Due to the choice of the test vector set, the error counts are identical between the individual and the grouped versions.

```
(Success rates shown)

add       100.00%   3152/3152
fma        97.60%   4277/4382
RESULT     98.61%   7429/7534

   GROUPED

basic     100.00%   3152/3152
extra      97.60%   4277/4382
```

Listing 3.25: *Output of `eval.py -e basic ex.log`*

`basic_near` All information provided by the `basic` module is also provided by `basic_near`, with one important difference: Only test vectors that test the roundTiesToEven rounding mode are regarded, since this denotes the default and most widely employed rounding mode.

`error_list` The evaluation function `error_list` prints a list of all errors that occurred during the test run. The format is identical to the output format of `IeeeCC754++ -vio`. This analysis module enables review of the exact errors without needing to open the corresponding log files (where additionally the errors need to be localised). Listing 3.26 shows heavily shortened example output for only three of the 105 errors.

```
[__] l.7479 s fma n 80800000 00800000 00000000 80000000 xu => - 00000000 xu  |
    sa 0
[__] l.7480 s fma z 80800000 00800000 00000000 80000000 xu => - 00000000 xu  |
    sa 0
[__] l.7481 s fma u 80800000 00800000 00000000 80000000 xu => - 00000000 xu  |
    sa 0
```

Listing 3.26: *Output of* `eval.py -e error_list ex.log`

`error_report` As can be seen in Listing 3.26, `IeeeCC754++` records the type of error that occurred when executing a test vector. The analysis module `error_report` groups all errors according to the error type and shows detailed information for which of the operations (and how many times) this type of error occurred. Furthermore, the letters that encode the error type are decoded and printed into the report for easy viewing. The output in Listing 3.27 shows that the 105 errors in the example log file are of the same type, namely the sign of the returned result being wrong.[8] As these errors occurred for the `fma` operation, an additional fma error was recorded. In addition to the error count (105 in this example), the sum of ulps that the correct floating-point value differed from the returned value is printed (0 in this example).

```
(Operations, ulps, error count shown)

sa      - fma error
        - Different sign
    fma         0    105
```

Listing 3.27: *Output of* `eval.py -e error_report ex.log`

`operation_report` The evaluation function `operation_report` shows basically the same information as `error_report`, but grouped according to the operations that produced at least one error. This enables easy assessment of the types of errors that were encountered for a specific operation and therefore complements the information provided by `error_report`. The corresponding example output can be found in Listing 3.28.

```
(Errors, ulps, error count shown)

fma
    sa          0    105
        a - fma error
        s - Different sign
```

Listing 3.28: *Output of* `eval.py -e operation_report ex.log`

[8] The presence of sign errors and the absence of errors in exponent or significand mean that the recorded errors are related to signed zeroes being returned with the wrong sign.

roundings The analysis modules explained so far focus on generating summaries and a detailed view on errors, but from a rather operation-centric point of view. The `roundings` evaluation function aggregates test results according to the tested rounding modes tested in order to provide a quick overview whether one of the rounding modes is implemented in a less IEEE-conforming manner than the other rounding modes or whether errors are distributed homogeneously between rounding modes.

The example output in Listing 3.29 shows quite interesting results in this regard: Whereas errors are spread uniformly between roundTiesToEven, roundTowardPositive, and roundTowardZero, no errors occurred at all for roundTowardNegative (for the vectors tested here). The letters that are used to describe the rounding modes are explained in Table B.2.

```
(Success rates shown)

n           98.78%   1859/1882
z           97.81%   1835/1876
u           97.83%   1847/1888
d          100.00%   1888/1888
RESULT      98.61%   7429/7534
```

Listing 3.29: *Output of* `eval.py -e roundings ex.log`

ulp As part of the analysis extensions, `IeeeCC754++` records the difference in ulps for test vectors where the returned value differs from the correct result, provided the returned result is close enough to the correct result (with a default threshold of 8 ulps, cf. Section 3.1.12). With this information, one can see which of the test vectors were in principle calculated correctly, but not properly rounded (1 ulp difference), which were calculated almost correctly (deviation between 2 and 8 ulps), and for which the returned result is completely wrong. The reasons for these cases are different: In the first case, the algorithm used for the computation of the result seems to be correct, but one or more rounding modes are not supported or not properly implemented, while in the second case, the algorithm does not provide the required accuracy. Finally, the third case can hint at more fundamental problems with the affected floating-point operators.

The `ulp` evaluation function retrieves the corresponding ulp values from the test results and displays the values for the three cases. Additionally, the number of executed test vectors is shown. In order to discern which rounding modes are affected by deviations in the significand, this information is generated for all tested rounding modes and is shown together with overall counts.

The example output in Listing 3.30 shows that cases with moderate deviations (greater than 1 ulp) only occur in seldom cases, whereas the case of the returned result being 1 ulp off is quite common. Note that the testset used to produce the output in Listing 3.30 is different from the testset used for the other examples in this section. It contains results for all test vectors (including the elementary

functions, cf. Chapter 4); the results were generated with the `c99` FPU of the `default` architecture (see Section 5.1.1) on an x86 platform with `gcc` 4.8.1. A deeper analysis reveals that most of the 1 ulp differences stem from test vectors for the elementary functions in rounding modes other than roundTiesToEven, indicating that the internal precision used to compute the elementary functions might not be high enough to round all cases correctly (probably due to the Table Maker's Dilemma, cf. Section 4.1.1). This assumption is strengthened by the counts for 1 ulp deviation being significantly larger for roundTowardPositive, roundTowardNegative, and roundTowardZero compared to roundTiesToEven.

This demonstrates that, when deviations in the returned results are found, this evaluation function can show the distribution of errors, but a more detailed analysis is needed to unveil the reasons for the deviations.

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n           23      0    194      8122
z          411      2    210      7664
u          547      8    201      7572
d          893      4    208      7585
RESULT    1874     14    813     30943
```

Listing 3.30: *Output of* `eval.py -e ulp ex2.log`

Aggregated evaluation functions

The evaluation functions described so far provide insight into one specific aspect of a test run (and thereby into the IEEE-conformity of the underlying floating-point environment). In order to gain a more comprehensive overview without having to perform several eval tasks, the evaluation framework provides three evaluation functions that do not add further analysis facilities, but aggregate some of the other evaluation functions.

The `detailed` function collects the output of the modules `basic`, `roundings`, `ulp`, `operation_report`, and `error_report` into one report. If additionally direct evaluation of the exact errors is needed, the `very_detailed` evaluation function can be used which aggregates the `detailed` and `error_list` functions.

Finally, the `all` module searches the `evalfunc/` directory for analysis modules and uses every evaluating function that was found to provide a report including the output of all functions. Note that in the default `IeeeCC754++` setup, the evaluation functions contained in the `detailed` module are printed three times (one time when executed on their own and again when the `detailed` and `very_detailed` functions are called). For similar reasons, the error list generated by `error_list` is printed twice. As this list tends to be quite long, the `all` evaluation function should only be used to check if all evaluation functions work as expected.

Adding custom analysis modules

The analysis modules supplied by `IeeeCC754++` cover a wide, but nonetheless basic range of analysis needs particularly helpful towards evaluating the basic floating-point conformity and properties of a given floating-point environment in a quick manner, something especially important when a large number of test runs needs to be evaluated. For more specific and elaborate analysis tasks, the evaluation framework can easily be extended by adding custom analysis modules. In this section, we give a short description of the steps required to implement such a module. For further details, we refer to the implementation of the `example` module implemented in the file `evalfunc/example.py` (see Figure 3.3, page 113). This file does not implement a working analysis module, but explains the basic steps necessary to retrieve the relevant data from the database, and performs example queries against the table containing the testing results. The (heavily commented) full source code can also be found in Appendix C.6.

All analysis modules are located in the directory `evalfunc/` (see above). The first step towards creating a custom module is to generate a new `Python` source file in that directory (or to copy one of the other analysis modules). In this short example, we name this file `minimal.py`

In order to transform this empty file into an evaluation module, two mandatory functions need to be implemented: `version()` that returns a string containing information about the current module, such as version information, and `evaluate(db)`. The latter function takes an `SQLite` database `db` as a parameter and returns a string containing the report of the evaluation function. Listing 3.31 contains a minimal implementation of an analysis module.

```
# Version information
def version():
    return "Minimal analysis module v0.01"

# The actual evaluation function
def evaluate(db):
    return "Empty evaluation report of the minimal analysis module."
```

Listing 3.31: *`minimal.py`: Minimal implementation of an analysis module.*

The first step towards verifying that the new module works as expected consists of ensuring that the evaluation framework properly detects the module:

```
[src]> cd eval/
[src/eval]> python eval.py --list
Looking for SQLite and pySQLite.
Try system install (sqlite3)... found SQLite v3.7.17 and pySQLite v2.6.0.
eval.py: IeeeCC754++ job analyser

Known eval functions:

all  =>  all v0.1
```

```
...
minimal  =>  Minimal analysis module v0.01
...
very_detailed  =>  very_detailed v0.2
```

Listing 3.32: *Listing the new analysis module.*

If the module is not shown in this list, it is not detected by the evaluation framework and can therefore not be executed. Most likely, either one of the two mandatory functions has not been implemented, or the module itself cannot be parsed properly. The easiest way to remove syntax errors consists of executing the module directly and analysing `Python`'s error messages:

```
[src/eval]> python evalfunc/minimal.py
  File "evalfunc/minimal.py", line 6
    def evaluate(db:
                   ^
SyntaxError: invalid syntax
```

Listing 3.33: *Searching for errors in the new analysis module.*

In the short example in Listing 3.33, a closing brace has been forgotten in the signature of `evaluate(db)`.

When the evaluation framework successfully lists the new analysis module, it can be tested in standalone mode with a log file generated by `IeeeCC754++`'s verbose mode:

```
[src/eval]> python eval.py -e minimal logfile.log
...
>>> 2016-07-18 19:13:09 [eval] Eval results:
Empty evaluation report of the minimal analysis module.
...
>>> 2016-07-18 19:13:09 [main] Done.
```

Listing 3.34: *Testing the new analysis module.*

When the standalone execution is successful and the evaluation function produces the desired report, the new module can be used inside the evaluation framework to help in the analysis of `IeeeCC754++`'s log files.

3.4.3 Tools for (semi-)automated testing

The evaluation framework is aimed at providing an efficient way of testing various user environments and configurations in an automated manner. However, when the number of different settings that should be considered is large (which almost inevitably occurs especially when checking different architectures, FPUs, and

compiler versions in one environment), a huge number of test task files is required (at least four for every combination of architecture and compiler). When these test files are available, it is easy to collect the corresponding job task files into one master job file (or a few at most) so that a run of the evaluation framework covering all the different test runs can be performed by a single `job.py` execution. Therefore, the challenge of efficient automated testing with `IeeeCC754++` and the evaluation framework lies in the efficient generation of task files.

In this section, we describe some valuable tools that assist the user with the mass-generation of task files as well as with the execution of the tests and the analysis of the resulting log files.

`genJobs.py`

In a typical automated testing setup towards checking the IEEE-conformity of a given user environment, only a limited number of parameters varies for one architecture, such as the FPUs, the compilers or compiler versions, and the testsets that should be regarded. Most of the other necessary parameters tend to be identical between different test runs. Due to this reasoning, the tool `genJobs.py` was developed to generate all needed task files for one architecture from a single input file specifying all relevant parameters. Of course, this approach still limits the number of variations able to be produced from such a single setup, i. e. it might be necessary to generate more than one input file for a single architecture in the following cases:

- Not all FPUs that should be regarded can be built at the same time, e. g. due to different compiler settings.
- Only some of the compiler versions that should be tested are compatible with some FPUs.
- Different compilers need different compiler switches.

For the first two cases, separate input files are necessary in order to test all combinations. For the third case, one or more environment scripts can be used to set up the environment in a suitable manner (cf. Section 3.4.1).

For a given input file, `genJobs.py` generates task files as follows:

- For every compiler specified, a compile task file is generated, as well as a job task file containing this compile task file and further test and eval task files.
- For every given testset, two test task files and two eval task files are generated and added to the job task file, together with mode tasks that execute `v` and `x` mode tasks. This results in the following entries in the job task file (with `<NAME>` denoting the name of the job that includes the architecture and compiler):

```
m v
t <NAME>-v
e <NAME>-v
m x
t <NAME>-x
e <NAME>-x
```

- Finally, one master job task file is generated for the architecture simply including all job task files for the different compilers.

Table 3.7 shows all parameters permissible inside a valid input file for `genJobs.py`. Note that the syntax rules are identical to those of the regular task files, see Section 3.4.1.

Parameter	Needed?	Default	Purpose
`_archname`	yes	—	Displayed name of the architecture.
`_arch`	yes	—	`IeeeCC754++` architecture name.
`_compilers`	yes	—	List of compilers.
`_fpu`	no	main	List of FPUs.
`_cores`	no	1	Number of cores for the build step.
`_bits`	no	—	32 bit, 64 bit, or native build.
`_cflags`	no	—	`C` compiler flags.
`_cppflags`	no	—	Preprocessor flags.
`_cxxflags`	no	—	`C++` compiler flags
`_ldflags`	no	—	Linker flags
`_libs`	no	—	Library flags.
`_env`	no	—	List of environment scripts.
`_envtemplate`	no	(*)	Template for the naming of environment scripts.
`_mod`	no	—	List of environment modules.
`_modtemplate`	no	(*)	Template for the naming of environment modules.
`_buildargs`	no	—	Additional switches passed to `configure`.
`_testargs`	no	—	Additional switches passed to `IeeeCC754++`.
`_batch`	no	—	Batch system used for job submission.
`_execprefix`	no	—	Execution prefix, e. g. `mpiexec` call.
`_mode`	no	main	Build mode (see compile task files).
`_testsets`	no	t1s, t1d	List of testsets to be used.
`_testsetstemplate`	no	`[t]`	Template for the naming of testset files.
`_evalfunc`	no	—	Evaluation function.

Table 3.7: *Parameters for `genJobs.py` input files.*

_arch denotes the architecture as used in IeeeCC754++, e. g. default or x86. However, since it is possible to have multiple input files for one architecture as discussed above and the task file names include the name of the architecture, the parameter _archname is provided which sets the displayed architecture name, i. e. it affects only the naming of the task files. As an example, when the x86 architecture should be tested with SSE and AVX FPUs (whose instructions can not be built at the same time), one input file could use _archname = x86-sse and the other _archname = x86-avx, while both input files would include _arch = x86 so IeeeCC754++ knows which architecture source tree should be used.

Most other parameters should be self-explanatory since their meaning is identical to their equivalents in the corresponding task files, but the testset and environment script handling requires further explanation. _testsets contains a list of testsets that will be used as testset parameters during IeeeCC754++ execution. The given values are then applied to a testset name template whose default is [t], i. e. [t] is replaced by one entry of the testset list. In the default configuration, this results in the testsets t1s and t1d to be used (cf. Table 3.7). If a different naming scheme should be used, it can be specified via _testsetstemplate. This parameter must include [t] which will be replaced with the entries in testsets. In the default setting, the names of the testsets are taken "as is", i. e. "_testsetstemplate = [t]". The default behaviour could also be achieved by setting "_testsetstemplate = t1[t]" and "_testsets = s d".

The approach taken to name the environment scripts is similar in the regard that if _envtemplate is not empty, the current compiler is substituted in this string, i. e. an entry for an environment script "ENV = switchcc [c]" is generated (with [c] being replaced by the current compiler). For every entry <MYENV> in _env, an additional line "ENV = <MYENV>" is generated. This means that when only the entries for environment files given in _env should be used (or no environment scripts at all), the environment template must explicitly be cleared by specifying "_envtemplate =" in the input file. Finally, if neither _env nor _envtemplate are given, only the default entry "ENV = switchcc [c]" is added to compile and test task files.

For finer grained control over the naming scheme in _envtemplate, the three variables shown in Table 3.8 are substituted. Note that we assume that the compiler is given in the form "[c] = [cn]-[cv]", with everything before the last "-" being the compiler name and everything after that the compiler version.

Parameter	Substitution
[c]	Full compiler name.
[cn]	Short compiler name.
[cv]	Compiler version.

Table 3.8: *Parameters substituted in environment and module templates.*

The same logic in the generation of environment script entries also applies to environment modules, the names of the corresponding input parameters being `_mod` and `_modtemplate`. By default, `genJobs.py` generates only one environment script entry "`ENV = switchcc [c]`", see above. If the compiler should be loaded via an environment module, specify a corresponding template via `_modtemplate` or set "`_modtemplate = default`" to use the default template. This setting causes a line "`MODULE = compiler/[cn]/[cv]`" to be written. Additional modules should be specified via `_mod`. If only a module entry should be used, set `_modtemplate` to an appropriate value and explicitly disable environment scripts by using "`_envtemplate =`".

The logic behind choosing environment scripts and modules can be summarised as follows:

- Only default environment script: No settings needed.
- Environment script with non-standard naming: Specify `_envtemplate`.
- Additional environment scripts: Specify in `_env`.
- Additional environment modules: Specify in `_mod`.
- Environment module to load compiler settings: Set "`_modtemplate = default`" (or explicitly set `_modtemplate` to desired template scheme).
- Only environment module for compiler: Set `_modtemplate` as described before and disable environment scripts by adding a line "`_envtemplate =`".

`updateJobs.sh`

The script `updateJobs.sh` is a more specialised tool that searches input files for `genJobs.py` in a specific location and calls `genJobs.py` for every input file that was found. Its usage is as follows:

```
[eval]> ./updateJobs.sh --help
Takes all files in host/arch/<HOST>/*.in and uses
genJobs.py to create task files in eval/<TARGETDIR>.

Usage is: updateJobs.sh [-d] <HOST> [<TARGETDIR>]
  When -d is given, delete all files in <TARGETDIR>.
  Default target path is current dir.
```

Listing 3.35: *Output of `updateJobs.sh -help`*

As described in Listing 3.35, the input files must have the file ending `.in` and be located in the directory `host/arch/<HOST>/` inside the `IeeeCC754++` source tree, with `<HOST>` being the only mandatory parameter. If `<TARGETDIR>` is not given, the task files are created in the current directory, otherwise the specified

directory is used for the output files. In order to clean up the <TARGERTDIR> before creating new files, the switch -d can be used.

Since specifying correct parameters is important to avoid deleting or overwriting files that should be kept, updateJobs.sh shows the chosen settings and waits for confirmation of the values before actually calling genJobs.py or deleting all files in the target directory. After this confirmation, updateJobs.sh changes into the target directory and passes every file with ending .in inside host/arch/<HOST>/ to genJobs.py. The latter then creates the desired task files inside that target directory.

For each input file, a master job task file called <_archname>.job is generated inside the target directory. To initiate an evaluation framework run for this architecture including all sub tasks (compile, test, and eval tasks), this file must be passed to job.py as follows:

```
> python job.py <TARGETDIR>/<_archname>.job
```

startTests.sh

The script startTests.sh takes the automation of evaluation framework runs one step further by providing a one-stop solution to execute test runs for all known architectures and to (re-)generate the corresponding task files. It keeps all task files in an appropriate subdirectory, and it collects the output of the evaluation framework (which is written onto the console by default) into separate log files for each architecture. Furthermore, the most relevant command line options that influence an evaluation framework run are mirrored from job.py and can therefore be used as parameters for startTests.sh.

startTests.sh needs some basic settings which are set either via environment variables or via settings in an environment script called mytests.local. The preferred method is via mytests.local which is imported at the beginning of a startTests.sh execution. If this file cannot be found, the script uses the corresponding settings from the environment and quits if not all three needed variables are set. Listing 3.36 shows the variables that need to be set in an example mytests.local file on a host testhost. The corresponding task and log files are kept in the directory testdir (via ARCH), and the evaluation framework will be run for the architectures default, x86, and x86-avx. Note that these are architectures in the sense of <_archname>, not in the sense of architecture implementations inside IeeeCC754++.

```
# local file that collects available tests for the current host/arch

HOST="testhost"
MYTESTS="default x86 x86-avx"
ARCH="testdir"
```

Listing 3.36: *Example of a* `mytests.local` *file.*

In order to explain the features supported by `startTests.sh`, we start by discussing its syntax:

```
[eval]> ./startTests.sh --help
Importing default arch/tests from mytests.local.
Usage is: startTests.sh [OPTS] [TEST [...]]
Usage is: startTests.sh --refresh
Usage is: startTests.sh --save
Usage is: startTests.sh --list
Usage is: startTests.sh --help

  --refresh  Update arch dependent files.
  --save     Save log files to log/YYYYMMDD_HHMMSS.
  --list     Lists supported tests.
  --help     Shows this message.

[OPTS] can be one of
  -c          Test in classic mode.
  -x          Test in verbose mode, ignoring exceptions
  -s          Show output of subprocesses.
  -i          Show information about IeeeCC754++ executables.
  -d <LEVEL>  Use debug level <LEVEL> (0-5).
  -v          Use "very_detailed" eval function and use -i.
```

Listing 3.37: *Output of `startTests.sh -help`*

If called without parameters, all tests defined in the variable `MYTESTS` are executed one after the other. Single tests out of these can be performed by simply specifying their names on the command line. A list of the available tests can be obtained by using the parameter `--list` as shown in Listing 3.38 for the `mytests.local` file in Listing 3.36.

```
[eval]> ./startTests.sh --list
Importing default arch/tests from mytests.local.
Supported tests: default x86 x86-avx
```

Listing 3.38: *Output of `startTests.sh -list`*

The following switches are mirrored from `job.py` in order to influence how the evaluation framework performs during the building, testing, and evaluation tasks:

- `-c`/`-x`: By default, the evaluation framework employs the verbose mode as default testing mode. Similar to `job.py`'s `-m` switch, `-x` and `-c` can be used to change the default testing mode to verbose mode without regarding exceptions (`-x`) or classic mode (`-c`). For a more detailed discussion of how the testing mode can be influenced, see Section 3.4.1.

- `-s`: If the output of the subprocesses started by the evaluation framework should be further analysed, especially the output of the configuration and execution stages, it can be written to the log files. This behaviour is enabled by the switch `-s`.

- `-i`: This switch enables the execution of an information script in the testing stage of the evaluation framework that prints information about the built `IeeeCC754++` executable into the log files.
- `-d`: If an evaluation framework run cannot be successfully finished and the log files do not contain enough information about the problem, or if an error inside the evaluation framework is suspected, the debug level can be raised, resulting in significantly more detailed output on the log files.
- `-v`: The switch `-v` does not switch the evaluation framework to verbose mode (as could be surmised from `-x`/`-c`), but enables the evaluation function `very_detailed` and switches on the option `-i` (see above). It serves as a means to quickly raise the verbosity of the evaluation framework analysis to a very detailed level, allowing for a deeper inspection without having to change task files.

For detailed information about the corresponding `job.py` switches and its default behaviour, see Section 3.4.1.

`checkErrors.sh`

When a large number of evaluation framework runs is performed via `startTests.sh`, it can be quite cumbersome to manually analyse the resulting logfiles and verify that the test runs were executed successfully. In order to enable a quick overview of this more technical side, the script `checkErrors.sh` searches the generated log files for errors and warnings indicating problems during the execution of the test runs.

As the script is tailored especially towards working together with `startTest.sh`, it relies on the same basic setup: It looks for the environment variable `ARCH` in the file `mytests.local` (if it is located in the current directory) or in the environment and searches for all occurrences of the words "`ERROR`", "`WARNING`", and a few more in all log files in `ARCH`, i. e. in `ARCH/*.log`.

If the switch `-o` is supplied, `checkErrors.sh` opens the log files in an editor for further analysis. The default setting for this editor is `vi`, but it can be changed by setting an appropiate value for the environment variable `IEEECC_EDITOR` such as `less`, `emacs`, or any other viewer/editor that accepts the file to be opened as the first command line parameter.

`IeeeCC754++LogViewer`

Since `checkErrors.sh`'s focus is mainly the retrieval of a quick overview of errors that might have happened during execution of the evaluation framework, such as tests not able to be compiled or files not found, it is not an ideal tool to analyse the resulting log files in deeper detail. `IeeeCC754++` provides a much more advanced application for the analysis of evaluation framework log files

called `IeeeCC754++LogViewer`. It is written in `Python2` with the toolkit `wxPython`, version 2.8, and is located in the `tools/` subdirectory of the `IeeeCC754++` source tree. `IeeeCC754++LogViewer` retrieves the environment variable `ARCH` either from `mytest.local` or from the environment and looks for all log files located inside this directory, similar to `checkErrors.sh`. Every log file is then parsed for `.log` and `.csv` files created during the evaluation framework execution, i. e. for `IeeeCC754++` output files, evaluation function reports, and the scanned output files in csv format. Afterwards, the main application is started, and all mentioned files can conveniently be viewed inside `IeeeCC754++LogViewer`.

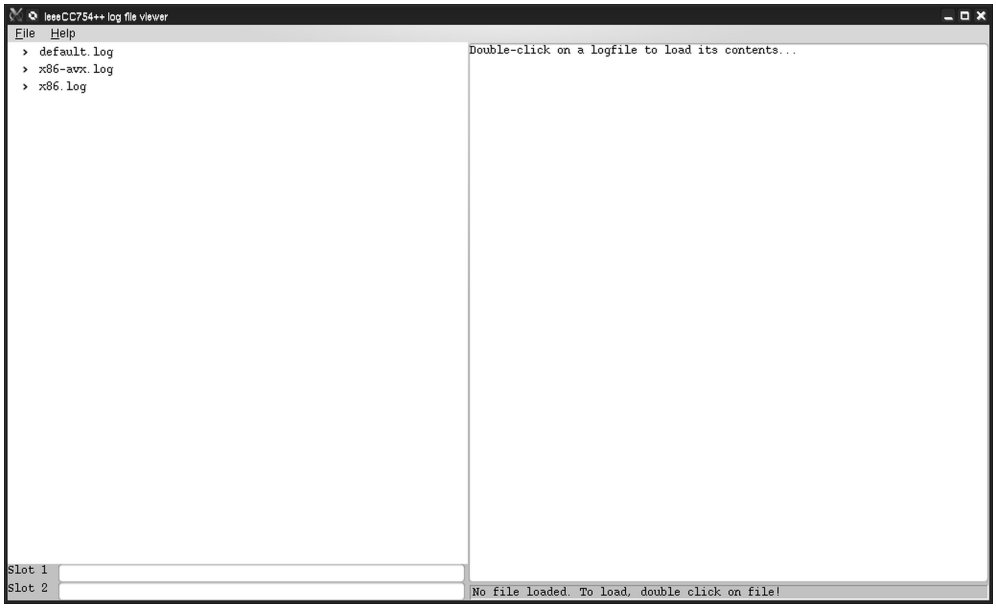

Figure 3.4: *`IeeeCC754++LogViewer` main window.*

Figure 3.4 shows the main window after the start of the viewer for the example tests shown in Listing 3.36. On the left, a tree-like structure shows the detected tests (or, more precisely, the detected log files generated by an evaluation framework run started via `startTests.sh`). Clicking the "`>`" sign expands the tree to show all files generated by the corresponding test run that generated the respective log file. Double clicking one of the entries in the tree displays the contents of that file in the right panel.

The files generated during a test run can be viewed inside that panel; navigating through these requires use of a mouse, either via the scroll wheel or the scroll bar on the right. For deeper inspection, all log files can be opened in an external editor by right-clicking on the file name and selecting "Open externally". This menu also offers a few further options: Choosing "Copy" puts the full path of the file into the clipboard, while "To Slot 1" and "To Slot 2" copy these file names into the corresponding slots in the lower left corner. Finally, "Display" has the same effect as double-clicking the file name, i. e. the file is displayed in the right

panel.

For the display and navigation of the "main" log files, an extensive set of keyboard shortcuts and further tools exist for jumping to the next marker which consist of the separators inserted by the evaluation functions, the section separators inserted via delimiter tasks, and the files generated by the evaluation framework. All shortcuts can be viewed by selecting the help dialog either via the "Help" item in the "Help" menu or via the shortcut "Ctrl + H". The generated files are displayed in blue and can be right-clicked, opening a context menu with basically the same options as discussed above.

The two slots in the lower left of the main window serve to collect the names of files which should be compared via an external compare/diff program such as Meld [Wil12], Diffuse [MM13], Kompare [FBS17], or Kdiff3 [Eib14]. Clicking on one of these slots opens a context menu with entries for comparing the two files (i. e. opening the files in the external diff program), opening the file in an external editor, pasting the file name into the clipboard, or clearing the corresponding slot.

Finally, the default choices for the font used in `IeeeCC754++LogViewer` as well as the external programs for viewing and comparing files can be changed via a settings file called `logviewer.conf` that must be located in the same directory as the `IeeeCC754++LogViewer` application itself. Listing 3.39 shows the allowed values together with the default values (which can be omitted, since they are built into `IeeeCC754++LogViewer`).

```
# set font
FONT = Adobe Courier
SIZE = 8

# set preferred editor and diff applications
EDITOR = kwrite
DIFF = meld
```

Listing 3.39: *logviewer.conf*

3.5 The optimisation framework

The choice of compiler crucially influences the performance of a user application in a number of areas, especially the choice of execution units, the runtime of the application, and its numerical IEEE-conformity. All of these parameters can be varied and controlled by switches supplied by the compiler. Unfortunately, there is an inherent tradeoff between faster execution and supporting full IEEE-conformity: For instance, support for (floating-point) exception handling involves checking for the occurrence of situations where exception flags should be raised. and thus imposes extra instructions to be performed. In some floating-point environments, support of subnormal numbers significantly slows down floating-point calculations

(cf. [MG14; HE02]). Therefore, disabling subnormal support might tremendously speed up the application.

The optimisation framework depicts a comprehensive attempt towards providing a tool for the evaluation of the influence of compiler options on both application performance (i. e. runtime) and floating-point conformity (i. e. IEEE-conformity). This is achieved by extending the evaluation framework with an *optimisation task* that executes all necessary `IeeeCC754++` runs from a single *opt task file*.

The rationale behind the optimisation framework is the following: Given a number of compiler options (specified in a hierarchical manner, see next section), it generates appropriate compile, test, and eval tasks. The eval tasks execute a specialised evaluation function that retrieves the results of the `IeeeCC754++` run and writes them into an optimisation database. After all tasks have been processed, one or more fitness functions are applied to the results in the database, and a "best" set of compiler options can be chosen for the user application.

While `IeeeCC754++` can be used to extract a user environment's floating-point conformity with regard to the chosen compiler and compiler options, it cannot be used to benchmark the target application's runtime performance. To cater for this need, the optimisation framework features facilities towards building the target application with the specified compiler settings and executing the resulting program. For maximum flexibility, the runtime can be recorded by the application itself (or some wrapper script) and be read from the application's output, or it can be measured by `IeeeCC754++`. Furthermore, it is possible to repeat the measurement process a few times in order to gain better statistics.

The process of setting up and executing an optimisation framework run is explained in Section 3.5.1. Afterwards, we explain how the optimisation framework can be extended with custom fitness functions, and finally, we discuss in detail the timing facilities and procedure used to measure application performance.

In order to use the optimisation framework, it is important to understand its limitations. These arise from `IeeeCC754++`'s approach to test IEEE-conformity itself: `IeeeCC754++` can only guarantee that (in case no errors are returned for a given testset) the tested operations are implemented in a manner conforming to IEEE 754-2008. This has two consequences: First, conformity can only be guaranteed for the specific test vectors executed during the testing phase, i. e. it is still possible that an operation will return e. g. an incorrectly rounded significand for a given combination of operands. Second, the evaluation of the effect of reordering operations (or whole loops) on the accuracy of an algorithm is out of `IeeeCC754++`'s scope. This means that, even when the floating-point environment is fully IEEE-conforming, the numerical results of an algorithm can change dramatically, depending on the environment, the algorithm and its stability, and the input data. An optimisation framework run (or, more precisely, an `IeeeCC754++` run) can never ensure optimal numerical results in some floating-point environment. However, it can assure the researcher that the environment's underlying floating-point implementation is performed in a thorough manner or

raise suspicions if too many errors are returned, and that under the specified compiler options relevant floating-point properties are not destroyed.

3.5.1 Using the optimisation framework

Since the optimisation framework is an extension of the evaluation framework, it shares the same configuration and execution philosophy. As a consequence, one run of the optimisation framework equals the execution of an opt task configured by an opt task file. This opt task can either be executed inside an evaluation framework run, or it can be executed standalone via `opt.py`.

When the optimisation framework is executed, it reads the given opt task file and generates compile, test, and eval tasks for every combination of compiler options. For performance reasons the optimisation framework does not generate task files, but uses in-memory data structures describing the respective tasks. This avoids the unnecessary creation of a huge number of task files. After executing the tasks for one combination, the target application is compiled with these compiler settings and executed a few times, and its runtime is recorded. Finally, the results need to be rated according to some criteria in order to choose an optimal set of compiler options. To achieve this, the optimisation framework provides fitness functions which assign each result a fitness value and sort the results according to the calculated values.

All parameters valid inside an opt task file are listed in Table 3.9, together with default values.

The meaning of most values is identical to those in the evaluation framework, see Section 3.4. In the following, we discuss the other parameters in greater detail:

- The optimisation framework allows for the evaluation of more than one compiler, specified via `COMPILERS`. Note however that all compilers listed here must accept all of the compiler switches given in the respective parameters.

- Since one result (or rather, one set of results) is needed for each set of compiler options in order to evaluate the floating-point conformity of the current floating-point environment, only one testset is allowed in `TESTSET`. This testset should comprise a sufficiently large selection of test vectors to enable a comprehensive evaluation of the environment's IEEE-conformity, such as the `t1` or `t3` testsets (cf. Listing 4.6).

- The compiler options that should be regarded during an optimisation framework run are specified in a hierarchical manner. This is due to the fact that some compiler options are mutually exclusive (such as `-O1`, `-O2`, etc. in `gcc`) and should not be combined. The optimisation framework allows for a maximum of three levels with every option on a lower level generating combinations with the options from the next level. The settings in Listing 3.40 result in the 8 combinations given in Listing 3.41.

Parameter	Mandatory?	Default	Purpose
ARCH	yes	—	IeeeCC754++ architecture.
ARCHNAME	no	ARCH	Displayed architecture name.
COMPILERS	yes	—	List of compilers.
MODULE	no	—	Environment module.
FPU	no	main	List of FPUs.
TESTSET	yes	—	Testset.
LEVEL1	no	—	List of compiler options, level 1.
LEVEL2	no	—	List of compiler options, level 2.
LEVEL3	no	—	List of compiler options, level 3.
COMBINATIONSLEVEL	no	0	How to apply combinations.
APP_BUILD	no	(*)	Architecture.
APP_EXEC	no	(*)	Architecture.
APP_REPEATS	no	1	Architecture.
APP_TIMING	no	internal	Architecture.
USE_EXTERNAL_APP	no	yes	Architecture.
FITNESS	no	success_rate	Architecture.
EVALFUNCTION	no	opteval	Architecture.
CFLAGS	no	—	Flags for the C-compiler.
CPPFLAGS	no	—	Flags for the preprocessor.
CXXFLAGS	no	—	Flags for the C++-compiler.
LDFLAGS	no	—	Flags for the linker.
LIBS	no	—	Extra libraries.
ARGS	no	—	Extra configure arguments.
CORES	no	1	Number of parallel builds.
MODE	no	main	Build mode.
BITS	no	—	32 or 64 bit build?

Table 3.9: *Compile task file parameters.*

```
LEVEL1 = -a
LEVEL2 = -b1 -b2
LEVEL3 = -c1 -c2
COMBINATIONSLEVEL = 0
```

Listing 3.40: *Example settings for compiler options.*

```
[no options]
-a
-a -b1
-a -b1 -c1
-a -b1 -c2
-a -b2
-a -b2 -c1
```

```
-a -b2 -c2
```

Listing 3.41: *Compiler option combinations resulting from Listing* 3.40.

Note that not all levels need to be defined, but if one level stays empty, the following levels are ignored. This means, if `LEVEL1` is empty, only one combination is tested, namely the one with empty options.

To provide for more flexibility in the generation of compiler option combinations, it is possible to test all combinations of options on levels 2 and 3. This behaviour is controlled by `COMBINATIONSLEVEL` according to Table 3.10. Listing 3.42 shows a variation of the example in Listing 3.40 with `COMBINATIONSLEVEL` set to 1, resulting in the combinations shown in Listing 3.43.

Value	Meaning
0	No combinations used (default behaviour).
1	Combinations used on the last level (either 2 or 3).
2	Combinations used from level 2 on, i. e. on levels 2 and 3).
3	Combinations used only on level 3.

Table 3.10: *Possible values for* `COMBINATIONSLEVEL`.

```
LEVEL1 = -a
LEVEL2 = -b1 -b2
LEVEL3 = -c1 -c2
COMBINATIONSLEVEL = 1
```

Listing 3.42: *Example settings for compiler options.*

```
[no options]
-a
-a -b1
-a -b1 -c1
-a -b1 -c2
-a -b1 -c1 -c2
-a -b2
-a -b2 -c1
-a -b2 -c2
-a -b2 -c1 -c2
```

Listing 3.43: *Compiler option combinations resulting from Listing* 3.42.

- In order to measure the runtime of the target application, the optimisation framework supports a number of variables for the specification of the necessary settings: `APP_BUILD` denotes a build script that controls the build behaviour of the target application, whereas `APP_EXEC` specifies an execution

script. APP_REPEATS controls how often the APP_EXEC script is repeated, and APP_TIMING can be set to either "internal" or "external" for different timing methods (runtime either measured by IeeeCC754++ or supplied by APP_EXEC). Finally, setting USE_EXTERNAL_APP to "no" can be used to disable the runtime measurement altogether. Details on these variables and the measurement procedure can be found in Section 3.5.3.

- Via FITNESS, it is possible to specify one or more fitness functions which will be applied to the results database at the end of an optimisation framework run. The fitness functions accompanying the optimisation framework are discussed in the next section, together with the process of extending the optimisation framework with custom fitness functions.

- Finally, EVALFUNCTION provides the possibility of specifying a custom evaluation function to be used inside the optimisation framework. However, this option should be used with extra care, since the default evaluation function opteval is specifically designed to extract results from IeeeCC754++ log files in a manner that these can be fed into the results database. Specifying a different evaluation function might break the optimisation framework's rating process. This parameter should therefore generally be avoided.

With these parameters, a (standalone) optimisation framework run can be started with the syntax shown in Listing 3.44.

```
> python opt.py --help
Looking for SQLite and pySQLite.
Try system install (sqlite3)... found SQLite v3.7.17 and pySQLite v2.6.0.
opt.py: IeeeCC754++ compiler options optimiser

Usage is:
  opt.py [-d <DEBUGLEVEL>] [-q] <FILE>
  opt.py -l|--list
  opt.py -h|--help

where
  -d <DEBUGLEVEL>   sets the level of debugging output:
                     -1 equivalent to -q [quiet]
                      0 normal
                      1 show "file found" messages
                      2 show path information
                      3 show info about tasks/actions etc.
                      4 full output: e.g. show content of generated scripts
  -q                quiet: suppress almost all opt.py output
  -l                list all known fitness functions
  -h                this message
  <FILE>            name an .opt task file whose contents will be executed
```

Listing 3.44: *Output of* `opt.py -help`

The only mandatory parameter is the opt task file <FILE> containing the settings for the current optimisation framework run. The parameters -d and

`-q` work exactly as in the evaluation framework (cf. Listing 3.16). Furthermore, `-l` shows a list of available fitness functions in a manner similar to the list of evaluation functions emitted by `eval.py -l` (see Listing 3.23).

3.5.2 Fitness modules and adding fitness functions

Similar to analysis modules in the evaluation framework (cf. Section 3.4.2), every fitness function known to the optimisation framework is contained in a fitness module. The process of selecting a "best" set of compiler options according to some criteria works as follows: First, the optimisation framework executes all necessary compile, test, and eval tasks, filling the optimisation database with the specially designed evaluation function `opteval` during execution of the eval tasks. `opteval` retrieves the results of an `IeeeCC754++` run, counts successes and errors, and pushes the following values into the optimisation database: overall count of test vectors, success and error counts (i. e. number of test vectors which returned a success or error), success and error rates, and success and error rates without regarding exceptions (cf. verbose mode `-vix`, Section 3.3.2). Additionally, the (average) runtime of the external application is recorded in the database. After all tests have been executed, all specified fitness modules are applied, i. e. the fitness function inside the fitness modules are called with the optimisation database as the only parameter. The fitness functions then evaluate all entries in the database, assign a fitness value according to the criteria implemented in the fitness function, and push that fitness value into the database. Finally, the optimisation framework sorts the optimisation database according to these fitness values and prints the resulting output table (higher fitness value meaning "better" set of compiler options according to the currently applied fitness function).

In this section, we describe the fitness modules contained in the optimisation framework. When the supplied fitness functions and their contained selection criteria are not sufficient to choose a "best" set of compiler options, it is possible to extend the optimisation framework with custom fitness functions. This process is also described later in this section.

An overview of the pre-installed fitness functions can be retrieved with the `--list` parameter of `opt.py`, see Listing 3.44; example output can be found in Listing 3.45. Figure 3.5 shows the files that implement the different evaluation functions as `Python` modules: `__init__.py` and `list_fitnessfuncs.py` initialise the `fitnessfunc` module and implement the listing functionality. `fitnesstools.py` provides common functionality able to be used in the fitness modules. Finally, all other files are named after a fitness function and contain the corresponding implementation, e. g. the file `runtime.py` contains the `runtime` fitness function.

```
python opt.py --list
Looking for SQLite and pySQLite.
Try system install (sqlite3)... found SQLite v3.7.17 and pySQLite v2.6.0.
opt.py: IeeeCC754++ compiler options optimiser
```

```
Known fitness functions:

error_rate  =>  error_rate fitness function v0.1
noexp_runtime  =>  success (without exceptions) + runtime fitness function v0.1
runtime  =>  runtime fitness function v0.1
runtime_noexp  =>  runtime + success (without exceptions) fitness function v0.1
runtime_success  =>  runtime + success fitness function v0.1
success_rate  =>  sucess_rate fitness function v0.1
success_runtime  =>  success + runtime fitness function v0.1
weighted  =>  weighted fitness function v0.1
```

Listing 3.45: *Output of* `opt.py -list`

```
eval/
└── fitnessfunc/
    ├── __init__.py
    ├── error_rate.py
    ├── example.py
    ├── fitnesstools.py
    ├── list_fitnessfuncs.py
    ├── noexp_runtime.py
    ├── runtime.py
    ├── runtime_noexp.py
    ├── runtime_success.py
    ├── success_rate.py
    ├── success_runtime.py
    └── weighted.py
```

Figure 3.5: *Code structure for the fitness modules inside the optimisation framework.*

Output of results

After applying a fitness function to the optimisation database at the end of an optimisation framework run, the resulting table is sorted according to the fitness value and printed onto the console. A plain text format is used for easy readability. The values displayed and their meaning are shown in Table 3.11.

An example is shown in Listing 3.47 for two compilers, two FPUs, and only one compiler option (to keep the output short). Note that in this example, no runtime information has been recorded. The exact settings used in the example are shown in Listing 3.46.

Available fitness functions

The supplied fitness functions focus mainly on two parameters: the application's runtime (where shorter runtime means better performance, which is usually

Parameter	Data type	Purpose
name	`string`	Unique name of current set of compiler options.
successrate	`float`	Success rate in percent.
errorrate	`float`	Error rate in percent.
noexp_rate	`float`	Success rate without regarding errors only related to exceptions.
runtime	`float`	Runtime of test application.
overall	`int`	Number of test vectors.
success	`int`	Number of test vectors without errors.
errors	`int`	Number of test vectors that returned errors.
noexp	`int`	Number of test vectors without errors, not regarding exceptions.
fitness	`float`	Fitness values computed by select fitness module.
arch	`string`	Architecture (`ARCHNAME`, see Table 3.9).
fpu	`string`	FPU.
compiler	`string`	Compiler.
compiler options	`string`	Current compiler options.

Table 3.11: *Entries in output table.*

```
ARCH = x86
COMPILERS = gcc-4.8
MODULE = compiler/[cn]/[cv]
FPU = main x87
TESTSET = t3d
LEVEL1 = -O3
USE_EXTERNAL_APP = no
FITNESS = success_rate
```

Listing 3.46: *Opt task file* `example_short.opt`.

highly desirable), and error or success rates with regard to floating-point accuracy (or, more specifically, IEEE-conformity). In order to not punish floating-point environments that do not support floating-point exceptions and to provide a means of ignoring exceptions, most modules exist in a variant where all errors reported by `IeeeCC754++` are counted as errors, and a second one where only errors related to the binary representation of the returned floating-point number are regarded.

The examples for the different fitness modules shown in this section were generated on an x86 platform with the settings shown in Listing 3.48. `sixloops` was used as external application, see Section 3.5.3. For detailed results for selected user environments, see Section 6.9.

Note that selecting a "best" set of compiler options for a specific target application is not a trivial task: In most cases, maximum performance is desired,

```
   success_rate

x86_gcc48_main_set00001   99.86   0.14 100.00    0.00  16661  16638     23
    16661 |  99.86 [x86 main gcc-4.8]
x86_gcc48_main_set00002   98.75   1.25 100.00    0.00  16661  16452    209
    16661 |  98.75 [x86 main gcc-4.8] -O3
x86_gcc48_x87_set00003    97.51   2.49  98.72    0.00  15483  15098    385
    15285 |  97.51 [x86 x87 gcc-4.8]
x86_gcc48_x87_set00004    97.51   2.49  98.72    0.00  15483  15098    385
    15285 |  97.51 [x86 x87 gcc-4.8]  -O3
```

Listing 3.47: *Output of* `opt.py example_short.opt`.

```
ARCH = x86
COMPILERS = gcc-4.7
MODULE = compiler/[cn]/[cv]
FPU = main
TESTSET = t3d
LEVEL1 = -O3
LEVEL2 = -funroll-loops -funsafe-math-optimizations
LEVEL3 =
COMBINATIONSLEVEL = 1
APP_BUILD   = sixloops_build.sh
APP_EXEC    = sixloops_execute.sh
APP_REPEATS = 3
APP_TIMING  = external
FITNESS = runtime success_rate error_rate runtime_success runtime_noexp
    success_runtime noexp_runtime weighted
```

Listing 3.48: *Example task file example.opt.*

i. e. shorter runtimes would be a strong criterion in the selection step. However, one has to carefully weigh the consequences and the severity of accepting too many floating-point errors in the application: Consider e. g. a floating-point environment where handling subnormal numbers imposes an increase in processor instructions that must be executed for every floating-point computation, and a compiler that provides a switch for turning off subnormal support. If using that compiler switch, the performance of the program might increase dramatically from a floating-point throughput point of view, but being much more inaccurate with numbers close to zero might either destroy results or lead to higher iteration counts in iterative numerical algorithms. Therefore, using numerical quality (i. e. high success rates) as the dominant criterion would in most cases lead to better numerical results.

`runtime` The `runtime` fitness function sorts the entries in the optimisation database according to their runtime. No further sorting criteria are used. The corresponding example is shown in Listing 3.49.

```
   runtime

x86_gcc47_main_set00008   73.86  26.14  78.42   15.33  16661  12305   4356
    13065 | 265.67 [x86 main gcc-4.7] -O3 -funroll-loops
    -funsafe-math-optimizations
x86_gcc47_main_set00006   98.75   1.25 100.00   15.41  16661  16452    209
    16661 | 265.59 [x86 main gcc-4.7] -O3 -funroll-loops
x86_gcc47_main_set00005   98.75   1.25 100.00   16.53  16661  16452    209
    16661 | 264.47 [x86 main gcc-4.7] -O3
x86_gcc47_main_set00007   73.86  26.14  78.42   16.62  16661  12305   4356
    13065 | 264.38 [x86 main gcc-4.7] -O3 -funsafe-math-optimizations
x86_gcc47_main_set00001   99.86   0.14 100.00  181.45  16661  16638     23
    16661 |  99.55 [x86 main gcc-4.7]
x86_gcc47_main_set00002   99.86   0.14 100.00  181.48  16661  16638     23
    16661 |  99.52 [x86 main gcc-4.7] -funroll-loops
x86_gcc47_main_set00003   73.31  26.69  77.27  181.70  16661  12215   4446
    12874 |  99.30 [x86 main gcc-4.7] -funsafe-math-optimizations
x86_gcc47_main_set00004   73.31  26.69  77.27  181.95  16661  12215   4446
    12874 |  99.05 [x86 main gcc-4.7] -funroll-loops -funsafe-math-optimizations
```

Listing 3.49: *Output of* `opt.py example.opt` *with* `runtime` *fitness function.*

`success_rate` Another pre-supplied fitness function that evaluates the generated results according to only a single criterion is called `success_rate`. This function serves as an example how to select the "best" combination of compiler options with regard to only one criterion already contained in the optimisation database. Since the success rate will usually be identical for typical sets of compiler options, due to the fact that most compiler options do not change floating-point semantics, this fitness function might not yield meaningful results in most cases. Listing 3.50 shows example output.

```
   success_rate

x86_gcc47_main_set00001   99.86   0.14 100.00  181.45  16661  16638     23
    16661 |  99.86 [x86 main gcc-4.7]
x86_gcc47_main_set00002   99.86   0.14 100.00  181.48  16661  16638     23
    16661 |  99.86 [x86 main gcc-4.7] -funroll-loops
x86_gcc47_main_set00005   98.75   1.25 100.00   16.53  16661  16452    209
    16661 |  98.75 [x86 main gcc-4.7] -O3
x86_gcc47_main_set00006   98.75   1.25 100.00   15.41  16661  16452    209
    16661 |  98.75 [x86 main gcc-4.7] -O3 -funroll-loops
x86_gcc47_main_set00007   73.86  26.14  78.42   16.62  16661  12305   4356
    13065 |  73.86 [x86 main gcc-4.7] -O3 -funsafe-math-optimizations
x86_gcc47_main_set00008   73.86  26.14  78.42   15.33  16661  12305   4356
    13065 |  73.86 [x86 main gcc-4.7] -O3 -funroll-loops
    -funsafe-math-optimizations
x86_gcc47_main_set00003   73.31  26.69  77.27  181.70  16661  12215   4446
    12874 |  73.31 [x86 main gcc-4.7] -funsafe-math-optimizations
x86_gcc47_main_set00004   73.31  26.69  77.27  181.95  16661  12215   4446
    12874 |  73.31 [x86 main gcc-4.7] -funroll-loops -funsafe-math-optimizations
```

Listing 3.50: *Example output with* `success_rate` *fitness function.*

error_rate The error_rate fitness function works exactly like success_rate, but uses the error rate instead of the success rate as the selection criterion. Since lower error rates are regarded as "better" than high error rates, the resulting output is always identical to that generated by the success_rate function.

runtime_success Even when the target application's runtime is chosen as dominant selection criterion (which in general is not a good idea, see above), it makes sense to use the quality of floating-point results as a secondary criterion.The evaluation function runtime_success implements this idea: It sorts the optimisation database according to runtime, and in case of identical runtimes favours the entry with higher success rate. Listing 3.51 demonstrates the effect on the example from Listing 3.48.

```
   runtime_success

x86_gcc47_main_set00008   73.86  26.14  78.42   15.33  16661  12305   4356
    13065 | 999.99 [x86 main gcc-4.7] -O3 -funroll-loops
    -funsafe-math-optimizations
x86_gcc47_main_set00006   98.75   1.25 100.00   15.41  16661  16452    209
    16661 | 999.98 [x86 main gcc-4.7] -O3 -funroll-loops
x86_gcc47_main_set00005   98.75   1.25 100.00   16.53  16661  16452    209
    16661 | 999.97 [x86 main gcc-4.7] -O3
x86_gcc47_main_set00007   73.86  26.14  78.42   16.62  16661  12305   4356
    13065 | 999.96 [x86 main gcc-4.7] -O3 -funsafe-math-optimizations
x86_gcc47_main_set00001   99.86   0.14 100.00  181.45  16661  16638     23
    16661 | 999.95 [x86 main gcc-4.7]
x86_gcc47_main_set00002   99.86   0.14 100.00  181.48  16661  16638     23
    16661 | 999.94 [x86 main gcc-4.7] -funroll-loops
x86_gcc47_main_set00003   73.31  26.69  77.27  181.70  16661  12215   4446
    12874 | 999.93 [x86 main gcc-4.7] -funsafe-math-optimizations
x86_gcc47_main_set00004   73.31  26.69  77.27  181.95  16661  12215   4446
    12874 | 999.92 [x86 main gcc-4.7] -funroll-loops -funsafe-math-optimizations
```

Listing 3.51: *Example output with runtime_success fitness function.*

Note that using the runtime as returned by the optimisation framework (even when averaged over some runs) does not constitute a particularly meaningful criterion when used as primary sorting parameter in conjunction with secondary parameters: Since the returned runtimes will be identical only in very few and highly unlikely cases, sorting according to this criterion will only consider the secondary criteria in those rare cases. As a remedy, an algorithm for clustering "similar" runtimes should be used as first selection criterion. However, this approach has not been implemented in the optimisation framework in favour of fitness functions better suited for this purpose, especially the weighted fitness function. The runtime_success fitness function therefore serves only as a simple implementation example.

runtime_noexp The fitness function runtime_noexp works in a manner similar to runtime_success, i. e. it uses the runtime as primary selection criterion and a floating-point success rate as secondary criterion. However, a test vector is

counted as success when the returned result is numerically correct, i. e. when sign, exponent, and significand have the correct value. In other words, all information regarding exception flags is ignored. Listing 3.52 shows the corresponding example.

```
   runtime_noexp

x86_gcc47_main_set00008   73.86  26.14  78.42   15.33  16661  12305   4356
    13065 | 999.99 [x86 main gcc-4.7] -O3 -funroll-loops
    -funsafe-math-optimizations
x86_gcc47_main_set00006   98.75   1.25 100.00   15.41  16661  16452    209
    16661 | 999.98 [x86 main gcc-4.7] -O3 -funroll-loops
x86_gcc47_main_set00005   98.75   1.25 100.00   16.53  16661  16452    209
    16661 | 999.97 [x86 main gcc-4.7] -O3
x86_gcc47_main_set00007   73.86  26.14  78.42   16.62  16661  12305   4356
    13065 | 999.96 [x86 main gcc-4.7] -O3 -funsafe-math-optimizations
x86_gcc47_main_set00001   99.86   0.14 100.00  181.45  16661  16638     23
    16661 | 999.95 [x86 main gcc-4.7]
x86_gcc47_main_set00002   99.86   0.14 100.00  181.48  16661  16638     23
    16661 | 999.94 [x86 main gcc-4.7] -funroll-loops
x86_gcc47_main_set00003   73.31  26.69  77.27  181.70  16661  12215   4446
    12874 | 999.93 [x86 main gcc-4.7] -funsafe-math-optimizations
x86_gcc47_main_set00004   73.31  26.69  77.27  181.95  16661  12215   4446
    12874 | 999.92 [x86 main gcc-4.7] -funroll-loops -funsafe-math-optimizations
```

Listing 3.52: *Example output with* `runtime_noexp` *fitness function.*

`success_runtime` and `noexp_runtime` If floating-point quality (in the sense of IEEE-conformity) is more desirable than absolutely shortest execution time, success rates should be used as dominant criterion when choosing the "best" set of compiler options. The optimisation framework provides two fitness functions executing this type of selection: `success_runtime` and `noexp_runtime`, the first using success rate as primary criterion and the second success rates without regarding errors only related to (floating-point) exception handling. In both fitness functions, the application's performance is used as second criterion. Listings 3.53 and 3.54 show corresponding examples.

```
   success_runtime

x86_gcc47_main_set00001   99.86   0.14 100.00  181.45  16661  16638     23
    16661 | 999.99 [x86 main gcc-4.7]
x86_gcc47_main_set00002   99.86   0.14 100.00  181.48  16661  16638     23
    16661 | 999.98 [x86 main gcc-4.7] -funroll-loops
x86_gcc47_main_set00006   98.75   1.25 100.00   15.41  16661  16452    209
    16661 | 999.97 [x86 main gcc-4.7] -O3 -funroll-loops
x86_gcc47_main_set00005   98.75   1.25 100.00   16.53  16661  16452    209
    16661 | 999.96 [x86 main gcc-4.7] -O3
x86_gcc47_main_set00008   73.86  26.14  78.42   15.33  16661  12305   4356
    13065 | 999.95 [x86 main gcc-4.7] -O3 -funroll-loops
    -funsafe-math-optimizations
x86_gcc47_main_set00007   73.86  26.14  78.42   16.62  16661  12305   4356
    13065 | 999.94 [x86 main gcc-4.7] -O3 -funsafe-math-optimizations
x86_gcc47_main_set00003   73.31  26.69  77.27  181.70  16661  12215   4446
    12874 | 999.93 [x86 main gcc-4.7] -funsafe-math-optimizations
```

```
x86_gcc47_main_set00004   73.31  26.69  77.27  181.95  16661  12215   4446
    12874 | 999.92 [x86 main gcc-4.7] -funroll-loops -funsafe-math-optimizations
```

Listing 3.53: *Example output with* `success_runtime` *fitness function.*

```
   noexp_runtime

x86_gcc47_main_set00006   98.75   1.25 100.00   15.41  16661  16452    209
    16661 | 999.99 [x86 main gcc-4.7] -O3 -funroll-loops
x86_gcc47_main_set00005   98.75   1.25 100.00   16.53  16661  16452    209
    16661 | 999.98 [x86 main gcc-4.7] -O3
x86_gcc47_main_set00001   99.86   0.14 100.00  181.45  16661  16638     23
    16661 | 999.97 [x86 main gcc-4.7]
x86_gcc47_main_set00002   99.86   0.14 100.00  181.48  16661  16638     23
    16661 | 999.96 [x86 main gcc-4.7] -funroll-loops
x86_gcc47_main_set00008   73.86  26.14  78.42   15.33  16661  12305   4356
    13065 | 999.95 [x86 main gcc-4.7] -O3 -funroll-loops
    -funsafe-math-optimizations
x86_gcc47_main_set00007   73.86  26.14  78.42   16.62  16661  12305   4356
    13065 | 999.94 [x86 main gcc-4.7] -O3 -funsafe-math-optimizations
x86_gcc47_main_set00003   73.31  26.69  77.27  181.70  16661  12215   4446
    12874 | 999.93 [x86 main gcc-4.7] -funsafe-math-optimizations
x86_gcc47_main_set00004   73.31  26.69  77.27  181.95  16661  12215   4446
    12874 | 999.92 [x86 main gcc-4.7] -funroll-loops -funsafe-math-optimizations
```

Listing 3.54: *Example output with* `noexp_runtime` *fitness function.*

`weighted` All fitness functions presented so far share a common trait: Their results can easily be generated by executing simple SQL statements to select and sort the data. For a more fine-grained selection of the "best" set of compiler options, the `weighted` fitness function can be used. The approach works as follows: For every entry in the optimisation database (i. e. for every set of compiler options and the corresponding results), four criteria are computed and normalised: the success rate, the success rate without regarding exceptions, the runtime, and the number of compiler options used. Each of these values is multiplied by a corresponding weight factor, and the fitness is then computed as the sum of these values.

With S the success rate, N the success rate without regarding exceptions, T the runtime of the external application, O the number of compiler options, and w_X the weight factor of criterion X with $w_X \in [0, 1]$, the fitness value f is computed as

$$f = w_S S + w_N N + w_T \left(1 - \frac{T}{T_{max}}\right) + w_O d_O \left(1 - \frac{O}{O_{max}}\right). \tag{3.1}$$

Note that T_{max} and O_{max} denote the maximum runtime and maximum number of compiler options (if either of these values is 0, it is set to 1 instead) and that d_O is an additional damping factor depending on O_{max}. Ideally, the sum of the

weights should be identical to 1. If one or more of these criteria should not be used, the corresponding weight can be set to 0.

The use of the number of compiler options as a selection criterion demands a short explanation: Whenever two sets of compiler options yield an identical fitness value, i. e. they are regarded to be equally "good" under the currently selected weight values, it is better to reach this result by applying fewer compiler options. To prevent this criterion from being too dominant, an additional damping factor d_O is used.

Listing 3.55 shows example output for the following weights: $w_S = 0.5$, $w_N = 0$, $w_T = 0.3$, $w_O = 0.2$, and $d_O = 0.5$.

```
   weighted

x86_gcc47_main_set00006   98.75   1.25 100.00   15.41  16661  16452    209
    16661 |  79.83 [x86 main gcc-4.7] -O3 -funroll-loops
x86_gcc47_main_set00005   98.75   1.25 100.00   16.53  16661  16452    209
    16661 |  79.65 [x86 main gcc-4.7] -O3
x86_gcc47_main_set00007   73.86  26.14  78.42   16.62  16661  12305   4356
    13065 |  67.19 [x86 main gcc-4.7] -O3 -funsafe-math-optimizations
x86_gcc47_main_set00008   73.86  26.14  78.42   15.33  16661  12305   4356
    13065 |  64.40 [x86 main gcc-4.7] -O3 -funroll-loops
    -funsafe-math-optimizations
x86_gcc47_main_set00001   99.86   0.14 100.00  181.45  16661  16638     23
    16661 |  53.01 [x86 main gcc-4.7]
x86_gcc47_main_set00002   99.86   0.14 100.00  181.48  16661  16638     23
    16661 |  53.01 [x86 main gcc-4.7] -funroll-loops
x86_gcc47_main_set00003   73.31  26.69  77.27  181.70  16661  12215   4446
    12874 |  39.70 [x86 main gcc-4.7] -funsafe-math-optimizations
x86_gcc47_main_set00004   73.31  26.69  77.27  181.95  16661  12215   4446
    12874 |  39.66 [x86 main gcc-4.7] -funroll-loops -funsafe-math-optimizations
```

Listing 3.55: *Example output with* `weighted` *fitness function.*

Adding custom fitness modules

The optimisation framework's built-in fitness functions comprises a set of very basic fitness functions, as well as a more elaborate weighted fitness function offering a much more general selection process. For most needs, it should be sufficient to edit this `weighted` fitness module implemented in `fitnessfunc/weighted.py` (see Figure 3.5, page 135) and adapt the weight factors for the desired effect.

However, in case a different selection process is desired, the optimisation framework can be easily extended by adding custom fitness modules. Since the process is similar to adding a custom analysis module to the evaluation framework, we refer to Section 3.4.2 for a general guideline. For details, the file `fitnessfunc/example.py` which implements the `example` fitness function can be used as reference. It contains a thorough description of the process of retrieving the necessary data, computing corresponding fitness values, and pushing them back into the optimisation database. The (heavily commented) full source code can also be found in Appendix C.7.

A custom fitness module `custom` must be implemented conforming to the following rules:

- The file implementing the module must be called identically to the module name, and it must be located in `eval/fitnessfunc/`. In other words, the `custom` module must be implemented inside `eval/fitnessfunc/custom.py`.
- `custom.py` must contain exactly two methods: `version()` which returns a string containing version information, and `fitness(db)` which takes the `SQLite` database as single parameter. The latter method should then retrieve the contents of the optimisation database, compute a fitness value for every entry, and push these values back into the database. For more details, cf. `example.py` (see above) and the implementation files of the supplied fitness modules in `eval/fitnessfunc/`.

3.5.3 Timing the effect of compiler options

In order to select a "best" set of compiler options for a given application, the optimisation framework can be used to study the effect of compiler options on floating-point accuracy and IEEE-conformity, and the most promising sets of compiler options can be applied while compiling the target (user) application in order to study the effect on application performance. However, performing this process manually can be cumbersome. Therefore, the optimisation framework provides facilities towards integrating the measurement of application performance into the framework.

This integration is achieved with the following approach: For every set of compiler options, the optimisation framework generates compile, test, and eval tasks to build and execute `IeeeCC754++` with the current compiler options and to retrieve the results from the `IeeeCC754++` run. Afterwards, the external application is built with the current compiler options and is run a few times to gain reliable statistics (up to 10 repeats, see below). The average of these runtimes is then recorded into the optimisation database and passed to the fitness modules for evaluation.

Timing approach

To retrieve the runtime of the program, two methods can be used: If `APP_TIMING` is set to internal, the optimisation framework records the time between triggering the execution of the application and the return (i. e. end) of the application run. While this approach is simple to implement, it may not be the best way to measure application performance, since for every execution of the application, the time needed to initialise the application (e. g. filling a test matrix with values) is measured together with the performance-critical parts of the application, thereby influencing the measurement. As a remedy, `APP_TIMING` can be set to external. The

optimisation framework then reads the output of the application and assumes that the last non-empty line of the output contains the application runtime as a string representation of a floating-point number. With this method, the application itself can decide when to trigger the performance measurement to record the runtime of performance-critical parts of the algorithm executed. In case the runtime cannot be retrieved (e. g. due to parsing errors), the internally recorded runtime is used as a fallback.

The optimisation framework must know how to build and execute the external application. This is achieved with build and execution scripts specified via `APP_BUILD` and `APP_EXEC`. These scripts must also accept the following environment variables and command line parameters:

- The compiler name is passed to the script as the first (and only) command line parameter.

- The compiler options are passed via the usual corresponding environment variables. In other words, the environment variables `CFLAGS`, `CPPFLAGS`, `CXXFLAGS`, `LDFLAGS`, and `LIBS` are set according to the settings in the opt task file, and the build and execute scripts must read and use these variables when compiling and running the target application.

Note that the scripts are responsible for setting up their build and execution environment, following the parameters passed by the optimisation framework. In particular, when the compiler settings need to be set up via an environment script or environment modules, the build and execute scripts must contain corresponding instructions.

Note that, depending on the target application's runtime, an optimisation framework run can take up a significant amount of time, especially when a significant number of compiler options is passed to the optimisation framework and a large number of application repeats is used to gain better statistics concerning the performance of the application. In order to mitigate the latter effect, the optimisation framework allows for a maximum of 10 repetitions for every set of compiler options.

A sensible way of cutting down the time needed to arrive at a "best" set of compiler options is to first feed all compiler options that seem sensible for the chosen target application into the optimisation framework, while disabling use of an external application by setting "`USE_EXTERNAL_APP = no`". With the information retrieved from such an optimisation framework run, a smaller selection of compiler options that yield good numerical results (i. e. a high degree of IEEE-conformity) can be used in another optimisation framework run, this time using the target application to record its performance.

External applications provided by the optimisation framework

The optimisation framework includes two applications which can be used to study the influence of compiler options on IEEE-conformity and application performance:

`sixloops` We implemented `sixloops` as a simple matrix-matrix multiplication on real matrices, done in a tiled manner with six loops to exploit caching. More precisely, it performs $C = A \cdot B$ with $A, B, C \in \mathbb{R}^{n \times n}$ with a block size of n_b with $n, n_b \in \mathbb{N}$ and $n_b \ll n$, using ikj-form. For more details, see e. g. [GV96].

Although `sixloops` is not representative of a "real world" application, the effects of some common command line options influencing caches, vectors units, and ordering of loops, such as `-O3` or `-funroll-loops` (for `gcc`), can be studied with this small example application.

The optimisation framework includes two sets of scripts to support building and executing an appropriate `sixloops` binary: The scripts `sixloops_build.sh` and `sixloops_execute.sh` compile and run `sixloops` using environment scripts, while the scripts `sixloops_mod_build.sh` and `sixloops_mod_execute.sh` set up compiler and further variables via environment modules, cf. Section 3.4.1, page 104.

HPCG As a second (more complex) application, the optimisation framework includes HPCG [DHP15] which "is an effort to create a new metric for ranking HPC systems. [...] HPCG is designed to exercise computational and data access patterns that more closely match a different and broad set of important applications, and to give incentive to computer system designers to invest in capabilities that will have impact on the collective performance of these applications" [HPCG]. HPCG is "based on a simple additive Schwarz, symmetric Gauss-Seidel preconditioned conjugate gradient solver. [...] HPCG generates a regular sparse linear system that is mathematically similar to a finite element, finite volume or finite difference discretisation of a three-dimensional heat diffusion equation on a semi-regular grid. The problem is solved using domain decomposition with a conjugate gradient method that uses an additive Schwarz preconditioner. Each subdomain is preconditioned using a symmetric Gauss-Seidel sweep." [DHP15]

If no "real-world" external (user) application is available, HPCG gives valuable insight into the performance of the user environment. This especially applies to multicore or multiprocessor platforms as HPCG is designed for large HPC installations comprising up to thousands of nodes and processors. For a thorough discussion why HPCG represents a suitable benchmark for real-world applications (especially in contrast to HPL, see [Pet+16]), cf. [DHP15].

To make use of HPCG and multicore facilities available in the user environment, the optimisation framework includes four build scripts and two execution scripts:

- `hpcg_build_serial.sh` builds a serial version of HPCG that is executed on one core on the current user environment via `hpcg_execute.sh`.

- `hpcg_build_omp.sh` builds a version parallelised with OpenMP. Once more, `hpcg_execute.sh` serves as execution script.
- The HPCG binaries produced with the scripts `hpcg_build_mpi.sh` and `hpcg_build_parallel.sh` are both executed via `hpcg_execute_mpi.sh` as they are both compiled with MPI support. `hpcg_build_parallel.sh` additionally builds HPCG with MPI and OpenMP support, making use of all available compute resources (regarding CPUs).

Note that these scripts make use of environment scripts to set up the compiler and additional variables (cf. Section 3.4.1, page 104). In order to support environment modules, another set of scripts exist whose names start with "`hpcg_mod_`" instead of "`hpcg_`", i. e. the modules equivalent of e. g. `hpcg_build_parallel.sh` is called `hpcg_mod_build_parallel.sh`.

Adding an external application to the optimisation framework

When adding a custom target application to the optimisation framework, the build and execution scripts supplied for `sixloops` and HPCG can be used as a starting point. Figure 3.6 shows the code structure for external applications inside the optimisation framework. Custom build and execute scripts must be added to their default location `eval/app/scripts/`. They must support the requirements listed above, especially reading the compiler name from the command line and using the values supplied via the relevant environment variables.

```
eval/
└── app/
    ├── hpcg/
    │   ├── README
    │   └── ...
    ├── scripts/
    │   ├── hpcg_build_mpi.sh
    │   ├── hpcg_build_omp.sh
    │   ├── hpcg_build_parallel.sh
    │   ├── hpcg_build_serial.sh
    │   ├── hpcg_execute.sh
    │   ├── hpcg_execute_mpi.sh
    │   ├── ...
    │   ├── sixloops_build.sh
    │   └── sixloops_execute.sh
    └── sixloops/
        ├── README
        └── sixloops.cc
```

Figure 3.6: *Code structure for external applications inside the optimisation framework.*

Chapter 4

Extended testsets

Since `IeeeCC754++` employs test vectors to check different aspects of a floating-point environment, the quality of the results depends on the quality and choice of these test vectors. For the operators supported by the original `IeeeCC754`, a sufficiently large number of test vectors was already available (cf. Chapter 2). In this chapter, we describe the operators that we added to `IeeeCC754++`, as well as the test vectors complementary to the `IeeeCC754` testsets in order to support the new operators. We start this chapter with some general considerations concerning the choice of operators added, as well as the difficulty of generating appropriate test vectors for certain operators (especially for the elementary functions). Afterwards, we give detailed descriptions of the new operators and test vectors. Finally, we present a tool to generate collections of test vectors (i. e. different variants of testsets) from the test vectors supplied with `IeeeCC754++`.

4.1 General considerations

Test vectors have to be chosen carefully in order to enable appropriate assessment of the IEEE-conformity of a given floating-point environment (cf. the discussion in Section 3.1.1). We follow the design decision already made with `IeeeCC754`: using test vectors for which it is difficult to achieve correctly rounded results. Furthermore, some simple test vectors checking fundamental operator functionality are provided as well as test vectors verifying correct behaviour concerning exceptions and special values such as NaNs and infinities. Finally, for some operators it is possible to reuse test vectors originally designed for other operators, such as using square root test vectors for the nth root $\sqrt[n]{x}$ (with $n = 2$) and nth power x^n (with $n = 0.5$) functions.

Ideally, the testsets should include test vectors especially challenging to round. However, for many floating-point operators, it is difficult to achieve worst cases due to the Table Maker's Dilemma.

4.1.1 The Table Maker's Dilemma

The *Table Maker's Dilemma* can be described as follows: Let $x \in \mathbb{F}^*(\beta, t, L, U)$ be a floating-point number, $f : \mathbb{R} \to \mathbb{R}$ a floating-point operator and $\circ : \mathbb{R} \to \mathbb{F}^*(\beta, t, L, U)$ one of the five rounding functions required by IEEE 754-2008. When computing $f(x) =: y \in \mathbb{R}$, we expect the resulting floating-point number $\circ(f(x)) =: \tilde{y} \in \mathbb{F}^*(\beta, t, L, U)$ to be correctly rounded, i. e. we expect the returned result to be the floating-point number closest to the correct (real) value y according to the current rounding mode, rounded to target precision t. For "simple" operators such as addition or multiplication, the intermediate precision m necessary for the calculation of this result is known beforehand (cf. e. g. [VCV01a; VCV01b]). However, for transcendental functions such as sine, exponentials or logarithms, it is difficult to achieve upper bounds on m. This is especially true if y lies very close to a *breakpoint*: either a floating-point number (for roundTowardPositive, roundTowardNegative, and roundTowardZero) or the exact middle between two adjacent floating-point numbers (for roundTiesToEven and roundTiesToAway, see e. g. [Mul$^+$10, Chapter 12]). In this case, the intermediate precision m needed to guarantee correct rounding can be significantly higher than for other inputs x. For performance reasons, it is highly desirable to use the smallest value for m sufficiently large enough to produce a correctly rounded result $\tilde{y}$. The problem of finding a suitable m depending on x and f is known as the Table Maker's Dilemma.

All floating-point numbers x that require the largest m to correctly compute $\tilde{y} = \circ(f(x))$ with target precision t constitute worst cases for the function f. If they (or cases that require slightly smaller m) are known, they can serve as ideal test vectors for the verification of correctly rounded results with `IeeeCC754++`. However, for some functions and a given floating-point format it is extremely difficult and time-consuming to compute these worst cases. We translated known results (as published e. g. in [LM01a], [LM01b], or [Mul$^+$10]) into test vectors in Coonen format for use with `IeeeCC754++`.

For a more detailed discussion on the Table Maker's Dilemma, see [LT98] and [Mul$^+$10, Chapter 12].

4.1.2 Adding operators and test vectors

Since one of the main goals of `IeeeCC754++` is to support IEEE 754-2008, we added all "recommended correctly rounded functions" mentioned in section 9.2 of IEEE 754-2008 [IEEE08]. Additionally, five operators were added belonging to the `C99` standard [C99]: the cube root, error functions, and gamma functions. For details, see Sections 4.3 and 4.6.

In the following sections, we give detailed explanations of the new operators supported by `IeeeCC754++` and the test vectors which have been added to the testsets in order to verify whether implementations support the new operators in

an IEEE-conforming way.

In this thesis, we often refer to groups of operators using the following terms:

Definition 4.1. We define categories for the floating-point operators supported by `IeeeCC754++` as follows (see also Section 1.2.1, page 11):

- *Basic operations*, also called *arithmetic operators*: addition, subtraction, multiplication, division, (floating-point) remainder, and square root, as well as fused multiply-add (`fma`).
- *Conversions*, which can be sub-categorised as follows:
  - Conversion between floating-point formats: round to format with smaller precision, convert to format with larger precision, and round to integral value.
  - Conversion from and to integer formats, i. e. conversion from a floating-point format to 32 or 64 bit integers or vice versa.
  - Conversion between binary and decimal values, i. e. rounding a binary floating-point number to the closest decimal number or rounding a decimal number to a floating-point number.
- *Elementary functions*: In this thesis, we denote all newly added operators (with the exception of `fma` which belongs to the basic operators) as elementary functions. These operators include trigonometric, exponential, logarithmic, power, and root functions, as well as error and gamma functions. ◊

Note that the term "conversion" is used in a loose sense here: It describes the process of transforming numbers from one format into another. As an example, conversion from one floating-point format into another floating-point format with larger precision is a real conversion (exponent and significand merely have to be rewritten), whereas conversion to a floating-point format with smaller precision might need to be rounded, thereby changing the original value in order to fit the value into the narrower format. The term "elementary" is also used in a loose sense: Traditionally, only trigonometric functions, exponentials and logarithms are regarded as elementary functions.

All new test vectors have been encoded in the precision independent Coonen format. This is especially relevant for all vectors checking special cases such as handling of infinities, NaNs, etc. However, research on worst cases for the elementary functions is mostly limited to double precision (e. g. in [LM01a], [LM01b], or [Mul+10]), and consequently, the corresponding test vectors only apply to double precision. They are encoded in Coonen format nonetheless in order to also be applicable when testing with `IeeeCC754++`'s classic mode (see Section 3.3.1).

4.2 Fused multiply-add

One of the most notable operator additions to `IeeeCC754++` is the fused multiply-add (`fma`) operator: It performs a multiplication followed by an addition with the intermediate result calculated with infinite precision.[1] It is noteworthy, since it is only rounded once (after performing both operations) and therefore can return results different from the results achieved by issuing separate operations (which would include a rounding step to t bits precision between the multiplication and the addition). `fma` was added to IEEE 754-2008 to reflect that many hardware FPUs already contained a `fma` instruction since scientific computations hugely benefit from a high-performing `fma`, which is typically used for operations of whole vectors of floating-point numbers such as `saxpy()` (cf. [LAP17]).

Table 4.1 shows the name of the function, the computed value, the name of the operator inside `IeeeCC754++`, and the signature of the corresponding function.

Name	Operation	`OP` name	Signature
`fma`	$x * y + z$	`OP::fma`	`T T::fma(T & y, T & z)`

Table 4.1: *The `fma` operation.*

4.2.1 Testset

Since $\texttt{fma}(x, y, z) = x * y + z$ includes a multiplication and an addition, all test vectors for multiplication and addition can be reused with $z = 0$ for multiplication and either $x = 1$ or $y = 1$ for addition (the latter being used in `IeeeCC754++` test vectors): In the first case, the multiplication is performed exactly as a regular multiplication, but without rounding. Adding $z = 0$ does change neither exponent, significand, nor exceptions, and the final rounding leads to the same result as if only the multiplication had been performed. Note that when the correct result is a negative zero -0 and the second operand in the original multiplication test vector is 0, the third operand has to be set to $z = -0$ to promote the signed zero.

For the addition case, the multiplication in the `fma` is performed with the first operand of the addition test vector and $y = 1$. The result of this multiplication is exactly the first operand without any changes, and no exceptions are raised. Performing the addition with z (the original second operand of the addition test vector) is then performed exactly as a regular floating-point addition.

This already large set of test vectors from the addition and multiplication testsets is complemented by new test vectors helpful towards distinguishing between implementations of the `fma` operator that are indeed fused, i. e. when only one final rounding is used, as well as implementations performing a regular floating-point

[1] Note that the intermediate precision is not indeed infinite, but a precision of $m = 3t + 5$ is sufficient to return a result correctly rounded to t bits (cf. e. g. [Mul+10, pp. 259-262]).

multiplication followed by a regular addition (with the intermediate result being rounded).

In the following example, we describe the principle used to build these test vectors. Note that not only this test vector is contained in the testsets, but also scaled variants with different exponents verifying that a fused `fma` is used. The example also demonstrates how test vectors are encoded in the precision independent Coonen and the precision dependent UCB format. In order to be applicable for arbitrary precision, the test vectors are stored in Coonen format:

```
Wfma    =       1i1     1i(2)2       -1i1       OK      1m1i1
```

Listing 4.1: *`fma` test vector in Coonen format.*

The test vector starts with a letter denoting the origin (in this case, "`W`" for our own test vectors) and the operator. It is followed by the specification of the rounding modes for which the test vector is valid, in this case roundTiesToEven mode, denoted by the "`=`" sign. The rest of the test vector contains the operands, the expected exceptions, and the correct result: The first operand x is the number 1 whose last bit in the significand is raised by 1 (denoted by "`i1`"), whereas the second operand is the number $y = 1.5$. The third operand is equal to the first operand, but with the sign bit set, i. e. $z = -x$. The "`OK`" denotes that no floating-point exception flags should be raised. Finally, the (correct) result in roundTiesToEven mode is almost equal to 0.5, i. e. 1 with exponent decreased by 1 ("`m1`"), but with the last bit in the significand set. For details on the exact syntax, see Listing 3.1.

Converting the test vector to UCB format with `IeeeCC754++` in conversion mode (cf. Section 3.3.1) results in the following line for single precision:

```
fmas n eq - 3f800001 3fc00000 bf800001 3f000001
```

Listing 4.2: *`fma` test vector in UCB format, single precision.*

The UCB test vector starts with the operator and the precision ("`s`"), followed by the rounding mode ("`n`" for roundTiesToEven) and a specifier making it clear that the result is to be checked for equality (i. e. the returned result is to be compared to the correct result, indicated by "`eq`"). The "`-`" sign specifies that no exception flags are expected. After these general specifications, the three operands and the correct result are given in hexadecimal notation. Details on UCB syntax can be found in Listing 3.2.

With these values, we can perform the multiplication and addition operations, as well as see the difference whether the intermediate step is rounded (or not). Multiplying x and y with infinite intermediate precision results in the intermediate result i_1 (the vertical line separates trailing bits not representable in single

precision):

$$\begin{aligned} i_1 = x * y = \;& 1.00000000000000000000001_2 \\ & * 1.1_2 \\ = \;& 1.10000000000000000000001|1_2 \end{aligned} \tag{4.1}$$

Still computing with infinite intermediate precision, adding z to i_1 yields the intermediate result i_2:

$$\begin{aligned} i_2 = i_1 + z = \;& 1.10000000000000000000001|1_2 \\ & - 1.00000000000000000000001_2 \\ = \;& 0.10000000000000000000000|1_2 \\ = \;& 1.00000000000000000000001_2 \cdot 2^{-1} \end{aligned} \tag{4.2}$$

Note that in the last step, the result has been normalised. The last bit in i_1 (which is not representable in single precision) is cancelled out when adding z, and therefore i_2 is representable in single precision, yielding the final result $r_{\texttt{fma}} = i_2$.

On the other hand, when computing $r = x * y + z$ with regular floating-point operations (called `mla` here for multiply-add), i. e. when rounding after computing $x * y$, the situation changes (here, $RN : \mathbb{R} \to \mathbb{F}^*(\beta, t, L, U)$ denotes the roundTiesToEven rounding function): The last bit has to be rounded upwards, resulting in

$$i_3 = RN(i_1) = 1.10000000000000000000010_2. \tag{4.3}$$

The final addition of z then yields

$$\begin{aligned} i_4 = i_3 + z = \;& 1.10000000000000000000010_2 \\ & - 1.00000000000000000000001_2 \\ = \;& 0.10000000000000000000001_2 \\ = \;& 1.00000000000000000000010_2 \cdot 2^{-1} \end{aligned} \tag{4.4}$$

After normalising, i_4 is again representable as a single precision number, so $r_{\texttt{mla}} = i_4$. Obviously, $r_{\texttt{fma}} \neq r_{\texttt{mla}}$. Consequently, when the test vector is executed in a floating-point environment, it can be deduced from the returned value whether the `fma` operator in that environment is indeed fused (i. e. no intermediate rounding is used when $r_{\texttt{fma}}$ is returned).

If neither $r_{\texttt{fma}}$ nor $r_{\texttt{mla}}$ are returned, it is possible that some even more serious problem concerning the computation of sums and products exists. In that case, the floating-point environment should be checked for other flaws with a deeper analysis using the other test vectors for addition and multiplication (and probably other operators).

A version of this example where the involved floating-point numbers are shown as IEEE 754-2008 single precision numbers in binary notation) can be found in Appendix C.1.

Using the principle demonstrated with this test vector, we added 38 precision independent test vectors to the `fma` testset aimed at verifying whether the `fma` implementation is indeed fused. These have been verified to be working correctly in single and double precision. Additionally, we added 22 more test vectors which only work for odd precision t, i. e. for precisions where the number of bits t used to represent the significand (including hidden bit) is odd. In particular, this means that the latter test vectors work in double precision (53 bit significand), but not in single precision (24 bit significand).

For debugging purposes, all of these 60 test vectors[2] that distinguish between `fma` and regular multiply-add are additionally contained in a test vector file called `mla`, together with 5 simple test vectors able to help verifying whether the order of input operands has been chosen correctly.[3]

4.2.2 Considerations concerning portability

The introduction of the `fma` operator into `IeeeCC754++` raises interesting questions concerning portability: Typical numerical programs involving inner products will contain sums of a large number of products. Therefore it is highly desirable to use an existing `fma` operator (especially if it is implemented in hardware) for performance reasons. However, this changes floating-point semantics, since these operations are computed with higher intermediate precision. In general, this will lead to better performing and more accurate computations, but the results will be different when executed in a floating-point environment lacking a `fma` operator. Note that this behaviour is similar to environments that compute intermediate results with higher precision, such as x86 platforms with x87 FPUs (where intermediate computations on double precision variables are computed with extended precision). The difference is that `fma` is by design not affected by double-rounding (cf. [Mul+10, pp. 75 sqq.] and [MMM13]).

4.3 Powers and roots

Table 4.2 lists the power and root functions added to `IeeeCC754++`. For the power function, three versions exist: the "normal" power function $\mathrm{pow}(x, y) = x^y$ with $x, y \in \mathbb{F}^*(\beta, t, L, U)$, a version with integral exponent $\mathrm{pow}(x, n) = x^n, n \in \mathbb{Z}$, and a variant with non-negative x, i. e. $x \in [0, \infty] \cap \mathbb{F}^*(\beta, t, L, U)$. The latter two versions were introduced by IEEE 754-2008, since they can be implemented more efficiently than the more general version `pow`.

[2] Note that these are Coonen test vectors, resulting in 300 UCB test vectors overall (one for each of the five IEEE 754-2008 rounding modes).

[3] `fma` assembler instructions may consist of three or four operands, with one operand (register) used as return value in four operand `fma`. As a result, the ordering of operands is not obvious in all cases.

In addition to the "pure" root and power operators (the first six rows in Table 4.2), two compound functions often used in geometrical or graphical computations were added: $\mathrm{comp}(x, n) = (1 + x)^n$ and $\mathrm{hypot}(x, y) = \sqrt{x^2 + y^2}$.

With the exception of the cube root (that is mandated by the `C99` standard), all other operators are recommended but not required by IEEE 754-2008. Note that all arguments are floating-point numbers, but arguments called n must be integral, i. e. $n \in \mathbb{Z}$.

Name	Operation	`OP` name	Signature
Roots	$\sqrt[3]{x}$	`OP::cbrt`	`T T::cbrt()`
	$1/\sqrt{x}$	`OP::rsqrt`	`T T::rsqrt()`
	$\sqrt[n]{x}$	`OP::rootn`	`T T::rootn(T & y)`
Power	x^y	`OP::pow`	`T T::pow(T & y)`
	x^n	`OP::pown`	`T T::pown(T & y)`
	$x^y, x > 0$	`OP::powr`	`T T::powr(T & y)`
Compound	$(1 + x)^n$	`OP::comp`	`T T::comp(T & y)`
Hypotenuse	$\sqrt{x^2 + y^2}$	`OP::hypot`	`T T::hypot(T & y)`

Table 4.2: *Power and root operations.*

4.3.1 Testsets

For the three power functions, some basic test cases were added, as well as a range of test vectors based on the special cases listed in section 9.2.1 of the IEEE 754-2008 standard. [Mul+10, p. 458] gives the worst case (i. e. hardest to round case, see Section 4.1.1) for the power functions in double precision. [Kor+10] gives another hard to round case for integer power functions in double precision. These have also been added as test vectors to the testsets for all three power functions. Additionally, the `pow` testset has been complemented by all test vectors from the square root testset with the second operand set to $y = 0.5$.

The `rootn` testset consists of basic simple cases, special cases from IEEE 754-2008, as well as all square root test cases with the second operand set to $y = 2$. The `rsqrt` testset contains test vectors which check exception handling and worst cases for single and double precision from [LM01a]. For the cube root, only simple test vectors checking basic functionality have been added.

Finally, the testsets for the `comp` and `hypot` operators consist of simple test cases and all special cases given in IEEE 754-2008.

4.4 Trigonometric functions

The IEEE 754-2008 standard recommends a large range of trigonometric functions that should be implemented by conforming floating-point environments: mainly

the basic functions sine, cosine, and tangent as well as their inverse operators, the hyperbolic trigonometric `sinh`, `cosh`, and `tanh` and their inverse operators. Furthermore, for some of these functions, versions were added that take multiples of π as arguments, such as `sinpi`, `cospi`, and `atanpi`. Note that some obvious candidates such as `asinpi`, `acospi`, or `tanpi` were omitted as they were not deemed important enough to be included in the standard.

The collection of trigonometric functions is rounded off by `atan2` and `atan2pi`: Both atan2pi(y, x) and atan2(y, x) describe the angle subtended at the origin by the point (x, y) and the positive x-axis. `atan2` measures angles in radians and its unbounded range is $[-\pi, \pi]$, whereas `atan2pi` measures angles in half-revolutions with a range of $[-1, 1]$ [IEEE08].

Table 4.3 shows a detailed list of the supported trigonometric operators and their signatures.

Name	Operation	`OP` name	Signature
Trigonometric functions	$\sin(x)$	`OP::sin`	`T T::sin()`
	$\cos(x)$	`OP::cos`	`T T::cos()`
	$\tan(x)$	`OP::tan`	`T T::tan()`
	$\text{atan2}(y, x)$	`OP::atan2`	`T T::atan2(T & y)`
	$\sin(\pi x)$	`OP::sinpi`	`T T::sinpi()`
	$\cos(\pi x)$	`OP::cospi`	`T T::cospi()`
	$\arctan(x)/\pi$	`OP::atanpi`	`T T::atanpi()`
	$\text{atan2}(y, x)/\pi$	`OP::atan2pi`	`T T::atan2pi(T & y)`
Inverse trigonometric functions	$\arcsin(x)$	`OP::asin`	`T T::asin()`
	$\arccos(x)$	`OP::acos`	`T T::acos()`
	$\arctan(x)$	`OP::atan`	`T T::atan()`
Hyperbolic functions	$\sinh(x)$	`OP::sin`	`T T::sinh()`
	$\cosh(x)$	`OP::cos`	`T T::cosh()`
	$\tanh(x)$	`OP::tan`	`T T::tanh()`
Inverse hyperbolic functions	$\text{arsinh}(x)$	`OP::asin`	`T T::asinh()`
	$\text{arcosh}(x)$	`OP::acos`	`T T::acosh()`
	$\text{artanh}(x)$	`OP::atan`	`T T::atanh()`

Table 4.3: *Trigonometric operations.*

4.4.1 Testsets

For the testsets for the trigonometric functions, there are three main sources: As usual, we encoded the special cases mentioned in section 9.2.1 of the IEEE 754-2008 standard as test vectors, providing full coverage of exception handling and corner cases. Additionally, all worst cases for the trigonometric functions described in [LM01b] and [Mul+10, pp. 451-457] are included in the `IeeeCC754++` testsets.

Finally, [Mul+10] gives input values for a number of functions for which no computation is necessary since the value is known beforehand, such as $\cos(x) = 1$ for small input values, i. e. $|x| < RN(\sqrt{2}) \times 2^{-27}$ (cf. [Mul+10, pp. 417-418]). From these relations (which are only valid for double precision), we generated corresponding test vectors for double precision.

Finally, some simple test vectors which check obvious values have been added.

4.5 Exponentials and logarithms

The IEEE 754-2008 standard recommends exponential and logarithmic operators for the most common bases, i. e. e, 2, and 10. Furthermore, it recommends variations of these functions that are in numerical computations: For exponentials, $b^x - 1$ is provided, whereas $\log_b(1 + x)$ should be supported for logarithms ($b \in \{e, 2, 10\}$). Table 4.4 shows the resulting functions.

Name	Operation	`OP` name	Signature
Exponentials	e^x	`OP::exp`	`T T::expx()`
	$e^x - 1$	`OP::expm1`	`T T::expm1()`
	2^x	`OP::exp2`	`T T::exp2()`
	$2^x - 1$	`OP::exp2m1`	`T T::exp2m1()`
	10^x	`OP::exp2`	`T T::exp10()`
	$10^x - 1$	`OP::exp10m1`	`T T::exp10m1()`
Logarithms	$\ln(x)$	`OP::log`	`T T::log()`
	$\log_2(x)$	`OP::log2`	`T T::log2()`
	$\log_{10}(x)$	`OP::log10`	`T T::log10()`
	$\ln(1 + x)$	`OP::logp1`	`T T::logp1()`
	$\log_2(1 + x)$	`OP::log2p1`	`T T::log2p1()`
	$\log_{10}(1 + x)$	`OP::log10p1`	`T T::log10p1()`

Table 4.4: *Exponential and logarithmic operations.*

4.5.1 Testsets

The test vectors used in the exponentials and logarithms testsets stem from the same sources as the trigonometric test vectors: Section 9.2.1 of the IEEE 754-2008 standard, [LM01b], [Mul+10, pp. 451-457], and [Mul+10, pp. 417-418]. Additionally, we added simple tests for easy cases, as well as vectors checking exceptions, overflows, and underflows. The vectors from [LM01b] and [Mul+10] are only valid in double precision, whereas all other test vectors are encoded for arbitrary precision.

4.6 Miscellaneous functions

The final four functions that are new in `IeeeCC754++` are not mentioned in the IEEE 754-2008 standard, but have been added to `IeeeCC754++` since they are required by the `C99` standard: the error functions `erf` and `erfc`, as well as the gamma functions `gam` and `lgam.`

The error function $\mathrm{erf}(x)$ and the complementary error function $\mathrm{erfc}(x)$ are defined as follows:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{4.5}$$

and

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt, \tag{4.6}$$

while the gamma functions $\mathrm{gam}(x)$ and $\mathrm{lgam}(x)$ are given as

$$\mathrm{gam}(x) = \Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \tag{4.7}$$

and

$$\mathrm{lgam}(x) = \ln |\Gamma(x)|. \tag{4.8}$$

The functions and their signatures are listed in Table 4.5.

Name	Operation	`OP` name	Signature
Error function	$\mathrm{erf}(x)$	`OP::erf`	`T T::erf()`
	$\mathrm{erfc}(x)$	`OP::erfc`	`T T::erfc()`
Gamma function	$\Gamma(x)$	`OP::gam`	`T T::gam()`
	$\ln \lvert\Gamma(x)\rvert$	`OP::lgam`	`T T::lgam()`

Table 4.5: *Miscellaneous operations.*

4.6.1 Testsets

To our knowledge, no literature on worst (or bad) cases for the four functions exist. As a consequence, the testsets contain only basic cases to verify basic functionality of the operators.

4.7 Generating testsets

The basic requirement for executing `IeeeCC754++` directly or via the evaluation framework is the availability of adequately prepared testsets in UCB format. For a maximum of flexibility concerning supported floating-point precisions, the basic `IeeeCC754++` testsets are encoded in Coonen format, which is precision independent

(cf. Section 3.1.11). In order to avoid cumbersome conversion by directly calling `IeeeCC754++` in conversion mode (cf. Section 3.3.1), `IeeeCC754++` comes with two tools to automate the conversion process and to generate collections of test vectors that target different sets of operations: `convertTestsets.py` converts test vectors from Coonen to UCB format, whereas `genUCB.sh` selects sets of operations from all generated UCB test vectors.

Before presenting these tools, we briefly discuss the choice of the precisions and the layout of the testsets used in this thesis.

4.7.1 A note on precisions

Most of the test vectors contained in the Coonen test vectors in the directory `src/testsets/` are encoded in precision independent format. The exception are test vectors denoting worst cases for a specific precision, such as those for worst cases of elementary functions described earlier in this chapter. In other words, most test vectors are applicable to arbitrary floating-point formats. However, in practice only a very limited number of floating-point formats is used: Virtually all floating-point environments support single and double precision, and many environments supply a larger format, usually double-extended (e. g. Intel) or quadruple (e. g. POWER) precision. Additionally, half precision is being added to platforms in order to support machine learning applications. As a result of this reasoning, we only support these five precisions: half, single, double, extended, and quadruple.

In order to provide well-arranged collections of testsets, the tools presented below contain test vectors for one of these five precisions. However, composing testsets in this manner poses a problem for operations working on operands of different precision, especially conversion from one floating-point format into another: These could be placed in the testsets of either of the involved precisions (e. g. conversions between single and double precision might belong either to the single precision or the double precision testsets). We avoid duplicating test vectors in multiple testsets by choosing the following convention: Test vectors with operands in different precisions are placed in the testset with the widest precision (i. e. with the largest exponent and significand range). This means that a testset for a specific precision includes only test vectors which convert to and from narrower precision. We give two examples: A testset for half precision contains no conversions between floating-point formats at all, while a double precision testset contains test vectors converting between half and double precision as well as between single and double precision.

4.7.2 `convertTestsets.py`

The helper tool `convertTestsets.py` is located in `src/testsets`, which is the same directory where the Coonen encoded testset files are stored. It can be called

with the following syntax:

```
Usage is:
  convertTestsets.py [--annotate] [--path <path>] [--exe <exe>] <type> <prec>
  convertTestsets.py --list
  convertTestsets.py --help

  Convert testsets from Coonen to UCB format for the given <type> and <prec>.
  File name for the output file will be <type><prec>

  Options:
    --annotate  use IeeeCC754++ -a to add line numbers to UCB file
    --path      path to directory containing the testset files (in Coonen format)
    --exe       name of IeeeCC754++ executable
    --list      list types & precisions
    --help      show this help
```

Listing 4.3: *Output of* `convertTestsets.py -help`

In almost most cases, `<type>` should be "`all`" in order to include test vectors for all operations into the new testset. If only a subset of operations is to be regarded, these should be selected via `genUCB.sh` (see below).

`<prec>` denotes the floating-point precision for which the new testset will be generated. All possible values for `<type>` and `<prec>` can be seen in Listing 4.4. The meaning for the precision specifiers is shown in Listing 4.6.

```
Types:      all basic conv elem fma
Precisions: h s d l q
```

Listing 4.4: *Output of* `convertTestsets.py -list`

When called with "`all`" and a precision letter, `convertTestsets.py` generates testset files called "`all<prec>`". These should be copied to the `testsets` directory[4].

Please note that `convertTestsets.py` relies on a compiled `IeeeCC754++` executable being available, either located in the same directory as `convertTestsets.py` itself and called "`IeeeCC754++`", or its location and name specified via the parameter `--exe`. The conversion mode `-o` (cf. Section 3.3.1) is contained in every `IeeeCC754++` regardless of the user environment it was built for. Therefore, it is sufficient to build `IeeeCC754++` with any available compiler in default without extra configuration options like this:

```
> configure
> make
> cp src/def/IeeeCC754++_default <TESTSETPATH>/IeeeCC754++
```

(with `<TESTSETPATH>` being the path where `convertTestsets.py` resides, i. e. `src/testsets`).

[4] `testsets` is the directory in which the evaluation framework looks for testsets.

4.7.3 genUCB.sh

When the quality of a given floating-point environment should be evaluated or analysed, all available test vectors for the precisions to be regarded would be incorporated into the analysis. However, this might not be desirable in all cases: Consider e. g. a comparison of several FPUs of some floating-point environment in which only one of the FPUs supports conversion between binary floating-point formats and decimal output (as is the case for most user environments since the `main` FPU usually supports all conversions whereas most other FPUs do not). In this case, excluding the test vectors applicable to these conversions results in more convenient direct comparison between the FPUs, since exactly the same test vectors are executed for all FPUs (leading e. g. to identical test vector counts).

The script `genUCB.sh` offers the possibility to select predefined sets of test vectors from the "full" testset files generated with `convertTestsets.py`. Furthermore, by fixing a naming convention for the filtered testsets, it is easy to see which type of test vectors were regarded during a specific `IeeeCC754++` run. Since the generated testsets are aimed at being used inside the evaluation framework, `genUCB.sh` is located in the directory where the evaluation framework searches for testsets, i. e. in `testsets`. Listing 4.5 shows the syntax of a `genUCB.sh` invocation. Note that `genUCB.sh` assumes the availability of testset files called `all<prec>` containing test vectors in UCB format for the selected precision.

```
genUCB.sh - select test vectors from an input testset file in UCB format.

Usage is: genUCB.sh <type> <prec>|<inputfile> [<round>]
          genUCB.sh --help
          genUCB.sh --list

where:
  type        operations in the output file
  prec        precision of the test vectors
  round       rounding mode used
  inputfile   name of an input file

When <prec> is given, genUCB.sh expects input from a file called "all<prec>".
When <inputfile> is given, input is taken from that file.

  --help shows this help
  --list lists possible values for <type>, <prec> and <round>.
```

Listing 4.5: *Output of `genUCB.sh -help`*

Listing 4.6 shows the possible values for `<type>`, `<prec>`, and `<round>` in order to specify the desired set of operations, precision, and an optional rounding mode. If no rounding mode is specified, test vectors for all five rounding modes are included.

For easy readability, the sets of operations are assigned distinctive characters hinting at the context of the currently selected testset. The sets are grouped as follows:

```
Format for output files: t<type><prec>[<round>]

with
  type    0  custom
          1  all
          2  basic + conv [B + C]
          3  basic + conv (fp + int) [B + F + I or 2 - D]
          4  trig + exp + log + pow + misc [T + E + L + P + M]
          A  arithmetic (+ - * / fma)
          B  basic (+ - * / rem sqrt fma)
          C  conv [F + D + I]
          D  conv (dec)
          F  conv (fp)
          I  conv (int)
          T  trig
          E  exp
          L  log
          Q  exp + log (E + L)
          P  pow
          M  misc

  prec    h  half (binary16)
          s  single (binary32)
          d  double (binary64)
          l  long (binary64 extended)
          q  quad (binary128)

  round   n  near (roundTiesToEven)
          u  up (roundTowardPositive)
          d  down (roundTowardNegative)
          z  zero (roundTowardZero)
          a  near, ties to away (roundTiesToAway)
```

Listing 4.6: *Output of `genUCB.sh -list`*

- Most testsets assigned a capital letter denote a set of operations that do not overlap with other testsets. The exception are group `A` which is a subset of `B`, group `C` which collects all conversions into one testset, and group `Q` which combines exponential and logarithmic operators (see below).

- The testset denoted by `B` collects the basic arithmetic operations addition, subtraction, multiplication, and division, together with the very common remainder and square root operations. Since `fma` uses only addition and multiplication, it is also placed in this testset.

  If only the "purely arithmetic" operations should be tested, the testset `A` can be used which includes all test vectors from testset `B`, with the exception of remainder and square root (but includes `fma`).

- The conversions required by IEEE 754-2008 have been categorised into three disjoint testsets: `F` consists of the conversions between different floating-point formats, `D` denotes conversions between binary and decimal formats, and `I`

collects all conversions between floating-point and integer formats, including the "round to integral" operation which rounds a floating-point value to an integer value and returns this value as a floating-point number.

If all three of these conversions should be tested, the testset `C` which contains all test vectors of the conversion testsets `F`, `D`, and `I` can be used.

Note that the testset `F` contains only conversion to "smaller" floating-point formats, i. e. floating-point conversions from the selected precision `<prec>` into other floating-point formats are only contained for precisions with shorter significand. In other words, the testset `tFh` contains no test vectors for floating-point conversions and is thus empty, whereas `tFq` contains test vectors to convert quadruple precision floating-point numbers into half, single, double, and double extended precision numbers.

- Test vectors for trigonometric, exponential, and logarithmic operations are placed in the testsets `T`, `E`, and `L`, respectively. The testset `P` contains all operations related to power and root functions, with the exception of the square root which is contained in testset `B`. Furthermore, a testset containing both exponentials and logarithms is available with the letter `Q`.

- All operations that have not been placed into one of the other "capital letter testsets" comprise the *miscellaneous* testset `M`. Currently, this testset consists only of operations computing the gamma and error functions (cf. Section 4.6).

- In addition to these (mostly) disjoint testsets, four testsets collecting larger sets of operations exist, denoted by single-digit numbers. They consist of sets of operations typically used as input to the evaluation framework.

  The first of these testsets (denoted by `1`) comprises all operations. When no rounding mode is specified, it is an almost identical copy of the input testset file `all<prec>` (comment lines added by `genUCB.sh` are different).

  Testset `2` exists to test the functionality that IEEE 754-2008 requires from a conforming implementation, namely the basic operations (testset `B`) and all conversions (testset `C`). Since typically not all FPUs support conversion between binary and decimal formats, `3` contains the same operations as `2` with the exception of the test vectors contained in testset `D`.

  Finally, testset `4` contains test vectors for the operations which IEEE 754-2008 recommends, but does not require to be implemented, i. e. the elementary functions. Thus, testset `4` contains all test vectors from the testsets `T`, `E`, `L`, `P`, and `M`.

- In order to distinguish between the testsets generated by `genUCB.sh`, thus consisting of well-defined sets of test vectors and testsets that contain a

custom collection of test vectors, the letter `0` should be used. If `0` is specified as the `<type>` specifier, `genUCB.sh` will emit a message that a *custom* set of test vectors must be built with means external to the script.

Table 4.6 gives a detailed overview of the mapping between testsets generated by `genUCB.sh` and the operations that are contained in these testsets.

Operation	A	B	C	D	F	I	T	E	L	Q	P	M	1	2	3	4
OP::add	×	×											×	×	×	
OP::sub	×	×											×	×	×	
OP::mul	×	×											×	×	×	
OP::div	×	×											×	×	×	
OP::rem		×											×	×	×	
OP::sqrt		×											×	×	×	
OP::ct			×		×								×	×	×	
OP::rt			×		×								×	×	×	
OP::b2d			×	×									×	×		
OP::d2b			×	×									×	×		
OP::i			×			×							×	×	×	
OP::ri			×			×							×	×	×	
OP::rI			×			×							×	×	×	
OP::ru			×			×							×	×	×	
OP::rU			×			×							×	×	×	
OP::ci			×			×							×	×	×	
OP::cI			×			×							×	×	×	
OP::cu			×			×							×	×	×	
OP::cU			×			×							×	×	×	
OP::fma	×	×											×	×	×	
OP::cbrt											×		×			×
OP::rootn											×		×			×
OP::pow											×		×			×
OP::pown											×		×			×
OP::powr											×		×			×
OP::sin							×						×			×
OP::cos							×						×			×
OP::tan							×						×			×
OP::sinpi							×						×			×
OP::cospi							×						×			×
OP::atanpi							×						×			×
OP::atan2pi							×						×			×
OP::asin							×						×			×

Continued on next page...

Operation	A	B	C	D	F	I	T	E	L	Q	P	M	1	2	3	4
OP::acos							×						×			×
OP::atan							×						×			×
OP::atan2							×						×			×
OP::sinh							×						×			×
OP::cosh							×						×			×
OP::tanh							×						×			×
OP::asinh							×						×			×
OP::acosh							×						×			×
OP::atanh							×						×			×
OP::exp								×		×			×			×
OP::expm1								×		×			×			×
OP::exp2								×		×			×			×
OP::exp2m1								×		×			×			×
OP::exp10								×		×			×			×
OP::exp10m1								×		×			×			×
OP::log									×	×			×			×
OP::log2									×	×			×			×
OP::log10									×	×			×			×
OP::logp1									×	×			×			×
OP::log2p1									×	×			×			×
OP::log10p1									×	×			×			×
OP::hypot											×		×			×
OP::rsqrt											×		×			×
OP::comp											×		×			×
OP::erf												×	×			×
OP::erfc												×	×			×
OP::gam												×	×			×
OP::lgam												×	×			×

Table 4.6: *Mapping between testsets and operations.*

Chapter 5

Architecture ports

The original `IeeeCC754` supported only a selected number of architecture ports and (logical) FPUs, namely generic x86 (both Intel and AMD) and SUN SPARC processors as well as the software libraries FMLib and MpIeee. This selection reflects the hardware platforms and software libraries available (and reasonably widespread at mathematical departments) during the time when `IeeeCC754` was originally developed.

Since then, the computing landscape has changed considerably, becoming much more diverse, and computing resources of all performance levels are available to a multitude of researchers. To reflect these changes and to cover a range of architectures as broad as possible, `IeeeCC754++` adds support for many of the most widespread architectures. Furthermore, it provides facilities to add further architectures in order to enable checking the IEEE-conformity of future architectures and architectures not yet implemented in `IeeeCC754++`.

This chapter is organised as follows: Each section is devoted either to a specific hardware architecture/ISA such as x86 (Section 5.2) or POWER (Section 5.4) or a class of similar user environments, e. g. GPUs and accelerators (Section 5.5) or virtual machines and software libraries (Section 5.7). In each section, we briefly describe the underlying architecture and its history. Afterwards, we examine in detail the corresponding ports that have been implemented as combinations of architectures and FPUs inside `IeeeCC754++`.

Before discussing the different platforms, it is important to note that an architecture port in `IeeeCC754++` sense abstracts from the underlying floating-point implementation (cf. Definition 3.3). As a result, an architecture port can be anything from a software floating-point library (such as MPFR, cf. Section 5.7.3, or SoftFloat, Section 5.7.2), the "standard" arithmetic operators in a programming language (e. g. `Java`, see Section 5.7.1), or the hardware implementation of an ISA (such as x86, Section 5.2). The last case is noteworthy insofar as the corresponding architecture ports in `IeeeCC754++` are implemented according to the ISA (especially when intrinsics or inline assembler implementations are used), but the compilation

process usually results in an executable for a specific hardware implementation of an ISA (which can still be generic or valid for a range of hardware), and testing is always performed on specific processors or arithmetic units. As an example, the x86 architecture port (see Section 5.2.1) is targeted at the x86 ISA [INT17b], which can then be compiled for e. g. current Intel Xeon CPUs. Finally, testing takes place on a specific processor out of the Xeon range, such as an Intel Xeon E5-2650 v4 [INT16c].

The architectures discussed in the next sections can be roughly categorised into the following groups: the most common hardware architectures such as processors with x86 ISA, platforms based on some variant of the POWER architecture (e. g. Cell and Blue Gene/Q), and ARM processors representing the most common architecture for smartphones, tablets etc.; some common accelerators such as GPUs (which are accessed through programming models such as CUDA and OpenCL); software libraries in the widest sense (MPFR, SoftFloat, CRlibm, and `Java` which is not software but virtual machine based), one architecture port specifically targeted at testing in-network floating-point computations (cf. Section 1.4.8); and finally the `default` architecture which enables analysing default floating-point environments, cf. Definition 3.1.

Note that this chapter is not aimed at giving an in-depth architecture description of the different platforms that are supported by `IeeeCC754++`. Although `IeeeCC754++` can help finding underlying hardware problems in specific FPUs, or even parts of an FPU like a floating-point pipeline, the goal of this thesis is to discuss the means via which a deep analysis is enabled, rather than looking at the hardware itself. A short overview of the respective platforms and their history is given. If details on specific architectures or FPUs are needed in order to understand a user environment's floating-point behaviour, we refer to the wealth of reference material available on the web.

In order to understand the architectures that are supplied with `IeeeCC754++` and what will actually be tested when the code for some FPU is executed, the following areas of the corresponding implementations are covered:

- Operators: The main differentiation between architectures is the way floating-point operators are supported in the respective user environment. If knowledge of the IEEE-conformity of the usual mathematical operators as supplied by `C++`, `C`, or the operating system as library function calls is desired, the `default` architecture should be used (or its relevant FPUs, see next section). In the other architectures, generally only those operators with specific hardware implementations are supported. For each architecture and FPU, we discuss which operators are implemented and why the specific implementation (such as an assembler or intrinsics version) has been chosen. We often refer to groups of operators such as basic operations or elementary functions; these are defined in Section 4.1.2.

- Rounding modes: Most of the architecture ports support the four rounding

modes required in IEEE 754, i. e. all rounding modes with the exception of roundTiesToAway (cf. Table 1.2). If an implementation deviates from this selection, we discuss the reasons and which rounding modes are supported. These four rounding modes are called the *classic rounding modes* in this chapter.

- Conversion to native format: Before executing a test vector, the floating-point values involved must be translated from `IeeeCC754++`'s internal floating-point format to the format(s) native to the corresponding architecture. In most cases, a conversion to `C++`'s floating-point formats is sufficient, taking endianess issues into account. The `default` architecture features an implementation supporting both little and big endian architectures. This implementation is also used in all architectures where the underlying platform supports both endianess variants, whereas architectures such as x86 provide only the variant used on that architecture (in that case little endian).

  When no special implementation is necessary for the conversion into the native format (and vice versa), either the generic version or the version targeted at a specific endianess is assumed. These functions are called *translation functions* throughout this section.

- Floating-point formats: Virtually all floating-point environments support the single and double formats, so in general, both formats are supported in all architectures and FPUs. Additionally, some environments support a larger floating-point format: either the double-extended format (x86) or the quadruple format (POWER). Currently, only the SoftFloat architecture provides an implementation for the half format (cf. Section 5.7.2). When no mention of supported formats is made, only single and double are supported.

- Environment handling functions: In order to set the correct rounding mode and to retrieve exception flags, a set of functions called `SetLibRound()`, `ClearLibExceptions()`, and `GetLibExceptions()` must be implemented, cf. Appendix B.4. For easy handling of the floating-point attributes, `C99` standardised a set of floating-point environment handling functions such as `fegetround()`/`fesetround()` or `feclearexcept`/`fegetexcept()` (which were usually already present before as library functions e. g. in the Linux operating system) in the header file `<fenv.h>`. Since most current `C++` compilers support this header file and the corresponding library, the standard approach of handling the operating system execution environment inside `IeeeCC754++` is an implementation making use of these functions.

  Only deviations from this approach are mentioned in the description of the architectures.

- Vector support: Some of the implemented FPUs support SIMD operation, i. e. executing the same operation simultaneously on vector operands. Vector

support in `IeeeCC754++` is discussed in Section 3.1.12 and Appendix B.7. In the following, only those FPUs that support SIMD operation are marked as vector units.

- Additional command line options: Some of the architectures support additional command line switches necessary for further setup, such as the accelerated architectures `nv`, `opencl`, and `mic` which provide for means to e. g. select between accelerated nodes. These options and their meaning are discussed for architectures where they are available.

Finally, we discuss the operators `b2d` and `d2b` (i. e. conversion between binary and decimal representation) which need special treatment: First, these conversions are (to our knowledge) always implemented in software, and the final code is almost always generated by the compiler of the chosen programming language. Consequently, they are usually not architecture dependent, so the canonical place to implement these functions is the `default main` FPU, since it serves as the basis for evaluating a user's default environment. In the other architectures, the `main` FPU is mostly a copy of this FPU. Therefore, `b2d` and `d2b` are copied over as well. However, for software floating-point environments that implement their own versions of these conversion operators, these are used instead of the generic versions in the `default main` FPU, such as in MPFR (see Section 5.7.3) and `Java` (see Section 5.7.1). Second, after using the conversion operators from binary to decimal representation, the resulting string must be transformed into a format that is parsable by `IeeeCC754++`. This postprocessing has been implemented into all versions of the conversion operators in the `main` FPUs. For details, see the implementation of `d2b()` in the `default main` FPU. As a last note, conversion between binary and decimal representations has only been implemented into the `main` FPU, so no other FPUs inside `IeeeCC754++` contain these operators.

Table 5.1 gives an overview of all architectures that have been implemented in `IeeeCC754++` as well as their respective FPUs.

5.1 The default architecture

We start the discussion of the different architectures and ports with two of the most basic variants: the `default` architecture which implements the default mode (cf. Sections 3.1.2 and 3.3.3) and the `dummy` architecture which serves as an implementation blueprint in case `IeeeCC754++` should be extended with a custom architecture (cf. also Appendix B).

5.1.1 The `default` port

The `default` architecture tries to mimic as far as possible the default user environment experienced by the user, cf. Definition 3.1. To achieve this goal, most

Architecture	Additional FPUs
default	generic, c99, cpp11, near
dummy	generic
x86	x87, 3dnow, sse, ssei, sses, sse3, sse3i, sse3s, avx, avxi, avxsse, avxssei, avx512, avx512i
mic	avx, avxi
arm	vfp, vfps, vfpv4, vfpv4s, neon, neonq, neoni, neonqi
aarch64	neoni, neonqi, asimd, sve
ppc	altivec, vsx, ppu, ppusimd, ppusimdi
cell	spu, spusimd, spusimdi
bgq	qpx, scalar
nv	cuda, cuda_rn, cudai
opencl	opencl, opencl_rn, opencl_round
mpi	
java	strict
softfloat	
mpfr	mpfrdef
crlibm	

Table 5.1: *Overview of the architectures and FPUs implemented in* `IeeeCC754++`*. Note that every architecture includes an implementation of the* `main` *FPU.*

of the operators are implemented without using any library function calls, solely relying on the means supplied by `C++` itself. The reasoning behind this approach was explained in Section 3.3.3.

The three basic floating-point formats defined in `C++` are usually available in most environments: **`float`**, **`double`**, and **`long double`**. However, the implementation of these formats may differ significantly. Whereas on x86 platforms **`long double`** is mostly mapped to Intel's 80 bit double-extended format (and would be executed in hardware inside the x87 FPU, see below), in other environments it might be mapped to a 128 bit quadruple format (which would most likely be executed in software since there are virtually no hardware FPUs for quadruple precision). In almost all cases, **`float`** is mapped to single precision and **`double`** to double precision.

All operations implemented inside the `default main` FPU support the three discussed `C++` floating-point formats. Note that due to the ambiguity of the mapping between **`long double`** and native formats, feeding quadruple test vectors to `IeeeCC754++` on an x86 platform will result in tons of errors, whereas using double-extended test vectors on e. g. POWER will lead to similarly erroneous behaviour. This means, that although the `default` architecture is aimed at being as general as possible, some knowledge of the underlying user environment

(cf. Definition 1.5) is needed in order to retrieve meaningful results for larger floating-point formats.

The basic operations are implemented using the usual `C++` operators, i. e. `+`, `-`, `*`, and `/`. Furthermore, most conversions are done using simple assignments to the new data type via the assignment operator `=` or using explicit `C`-style casts such as "`float a = (float)b`".

For conversions and operators where this approach is not possible, the library functions from `<cmath>` are used, such as `sqrt()`, `remainder()`, or `rint()`.

Note that the `main` FPU implements a "fake" `fma` operator, i. e. it simulates a `fma` operation by using `a * b + c` to compute $\mathrm{FMA}(a, b, c)$ with regular floating-point multiplications and additions. However, since this operator produces a significant number of errors during an `IeeeCC754++` test run due to the intermediate rounding (cf. Section 4.2), it is deactivated inside the `main` FPU. The `default generic` FPU (see below) enables this multiply-add implementation to support testing of this kind of "fake" `fma` operator, e. g. to check if a compiler combines multiplications and additions into `fma` operations (or to create new test vectors). In order to test a real `fma` operator (if available in the current user environment), the `c99` or `cpp11` FPUs can be used which call the `fma` implemented in the respective programming language's standard library.

Finally, one caveat concerning rounding modes should be mentioned: As roundTiesToAway is not required by IEEE 754-2008 as a rounding mode for binary floating-point implementations, most languages and operating systems do not support this rounding mode. Therefore, even when an underlying environment provides for roundTiesToAway, it is not possible to target this rounding mode via the `default` architecture, as this would break compatibility with almost all other environments. If roundTiesToAway support is necessary for some floating-point environment, one of the following implementation variants can be used:

- Adding a new architecture to `IeeeCC754++`. This is the preferred variant, although it might be too laborious if only the additional rounding mode is necessary. For details on how to add a new architecture, see Appendix B.

- Adding a new FPU to the `default` architecture. As described in Section 5.1.1, all operators can simply be forwarded to those of the `main` FPU. In order to support roundTiesToAway, only the function `SetLibRound()` must be overloaded (and all rounding modes registered inside `Register()` by calling `registerRDallIEEE()`), cf. Appendix B.4 and B.8.

- Changing the `default` architecture itself: Adding roundTiesToAway to `SetLibRound()` and registering the rounding mode can also be done to the `main` FPU of the `default` architecture. However, this approach is not recommended since the behaviour of the `default` architecture is changed, having an impact on the execution of `IeeeCC754++` in other floating-point

environments. Therefore, using one of the former variants is considered a cleaner approach.

The near FPU

When using IeeeCC754++'s default mode in floating-point environments in which only the default rounding mode roundTiesToEven is supported, IeeeCC754++ will detect incorrectly rounded results for some of the test vectors that check the other rounding modes. This is to be expected; nonetheless, in order to enable a clear view on actual errors in the underlying floating-point environment, IeeeCC754++ implements the near FPU that inherits all operators from the main FPU, but registers only roundTiesToEven as rounding mode.

The c99, cpp11 FPUs

In addition to the default user environment support implemented in the main FPU, IeeeCC754++ supports two FPUs which use the mathematical operators as supplied by the languages C++ and C themselves. For the cpp11 FPU (which implements C++11 support), this results in the operators looking almost identical to those in the main FPU, except that the functions from <cmath> are called via explicit scoping, i. e. by calling e. g. the square root function as std::sqrt() (or as ::sqrt for those compilers that put the corresponding functions into the global namespace instead of the namespace std).

The elementary functions like trigonometric functions, exponentials or logarithms were standardised in C99. However, the same functions from the C standard library have only been added to the official C++ standard with C++17, cf. [BNS16]. Since most compilers have been supporting these functions since C++11 nonetheless,[1] we provide implementations for those functions in the cpp11 FPU. The rare case of a compiler not supporting these functions can be recognised easily since compilation of the cpp11 FPU would fail.

The implementation of the c99 FPU follows the same philosophy as the main FPU in that it makes use of the operators built into C and otherwise calls the respective library functions. In order to actually test the floating-point behaviour of the C compiler, the implementation of the operators call functions declared as "**extern** "*C*"" and implemented in C source files. Inside these C files (which are compiled by the C compiler declared in MYCC, cf. Table A.2), the mathematical functions declared in <math.h> and required in the C99 standard are called.

Both FPUs support the **long double** format.

Note that both the c99 and cpp11 include the most comprehensive set of implementations for the mathematical operators supported by IeeeCC754++ (compared with the other architecture ports and their FPUs): basic operations (including

[1] The decision to not include the elementary mathematical functions unconditionally into the standard library was made during the C++11 standardisation process, see [BNS16].

remainder, square root, and `fma`), conversions (with the exception of conversions from and to decimal representation, cf. the note on page 168), and elementary functions, including those defined in `C99`, but not in IEEE 754-2008 (namely cube root, error functions, and gamma functions, see Section 4.1.2). However, note that a quite substantial number of elementary functions recommended by IEEE 754-2008 are missing from `C99`: the exponential and logarithmic functions `exp10`, `exp2m1`, `exp10m1`, `log2p1`, and `log10p1`, the reciprocal square root and nth root, the compound function, the `pown` and `powr` operators that compute powers for integer and positive exponents, and some trigonometric functions that take multiples of π as arguments, i. e. `sinPi`, `cosPi`, `atanPi`, and `atan2Pi` (for details on all these functions and operators, cf. Sections 4.2 to 4.6).

The `generic` FPU

The `generic` FPU serves as an example of how to implement a copy of another FPU. Since all FPUs in an architecture are inherited (either directly from the `main` FPU or one of the derived FPUs, see Appendix B.5), this can be achieved by forwarding all operations inside that FPU to the implementations of the FPU it was inherited from. Here, all operators inside the `generic` FPU simply call the operators defined in the `main` FPU. In addition to this, only the relevant conversion constructors need to be implemented, see Appendix B.5.

Since the `generic` FPU inherits all operators from the `main` FPU, it behaves identically, with one notable exception: The "fake" `fma` operator which is deactivated in the `main` FPU is available for testing in the `generic` FPU, e. g. for comparison purposes during the development of new `fma` test vectors.

5.1.2 The `dummy` port

One of `IeeeCC754++`'s main features is the possibility to extend the program with custom architectures. The `dummy` architecture serves as a blueprint for adding such an architecture. As such, its `main` FPU registers no operations at all, none of the functions handling environmental setup (rounding modes and exception flags) are implemented, and none of the conversion functions to translate floating-point values from `IeeeCC754++`'s internal format to a native format. Furthermore, there exists only a commented implementation of the addition operator for demonstration purposes.

For details on how the `dummy` architecture and its generic FPU are implemented to support extension with a custom architecture, see Appendix B

The `generic` FPU

The implementation of the `generic` FPU follows the same philosophy as the `generic` FPU in the `default` architecture, albeit with an even more basic approach:

Only the conversion constructors needed to translate between internal and native floating-point formats are implemented.

5.2 The x86 architecture

The x86 architecture represents probably the most widespread computer architecture of all time with regard to desktop and server computing systems. Its ancestry reaches back to 1978 when Intel introduced the 8086 processor. Since then, the x86 instruction set has been significantly increased with different additions and extensions, albeit in a manner that preserves backwards-compatibility [WIK17aj]. Although commonly associated with Intel, the x86 architecture has also been implemented in processors from Cyrix, AMD, VIA, and other companies, with AMD being the only current (as of 2017) competitor for Intel.

The modern x86 lineage started in 1985 with the introduction of the IA-32 architecture [INT03] which was first implemented in Intel's 80386 processor. In 2003, AMD introduced its Athlon 64 processor which implemented a 64 bit extension to the IA-32 instruction set called x86-64 [AMD05]. Since then, virtually all consumer PCs and a significant fraction of servers world-wide feature processors from Intel or AMD implementing this architecture. In this thesis, we summarise IA-32 and x86-64 under the term x86. For more details on the current x86 instruction set, see [INT17b].

To cater to the needs of current computing and software trends, a number of extensions to the x86 architecture have been introduced, such as extensions for multimedia processing (MMX, 3DNow!, SSE, AVX), virtualisation, or cryptography. From a floating-point arithmetic point of view, the most relevant features of the x86 architecture are the x87 coprocessor as well as the newer SIMD vector units called SSE and AVX. To check different aspects of these extensions, a number of FPUs have been implemented inside the `x86` architecture. The FPUs and their implementation inside `IeeeCC754++` are described below.

Another noteworthy variant of the x86 architecture is the Intel MIC (Many Integrated Cores) microarchitecture which was first released in 2012 with the Xeon Phi KNC (Knights Corner) chips, marketed by Intel as Xeon Phi x100 product family [INT15]. It is a manycore design based on the x86 ISA, with processors having 57 to 61 cores. In order to reach high floating-point throughput, each core has a 512 bit wide vector SIMD unit with an ISA called Intel ICMI (Initial Many Core Instructions) which is based on AVX and thus similar to a 512 bit variant of AVX2. KNC processors were only released as accelerators, i. e. as extension boards that supplement a host processor. It allows for two programming models: a native model where code is compiled for direct execution on a KNC processor, and an offload model where only certain parts of an application running on the host CPU are offloaded to the KNC and executed on the device.

The next incarnation of the MIC architecture was released in 2016 as Xeon

Phi x200 family of processors [INT17d], with 64 to 72 cores with an x86 compatible ISA having two AVX-512 SIMD units (see Section 5.2.1). These KNL chips were released initially as processors in a bootable form-factor, with an accelerator version being announced. However, in August 2017, Intel announced the discontinuation of the KNL coprocessor cards, resulting in KNL not being available in accelerator form [Shi17].

Support for KNL and KNC is described in Section 5.2.2.

5.2.1 The `x86` port

Since x86 is a little endian based architecture, only a little endian version of the translation functions is implemented. The environment handling functions make use of the library functions defined in `<fenv.h>`. One peculiarity of the x86 design needs to be mentioned here: Until SSE became widespread enough for compiler vendors to use SSE as the standard floating-point execution unit, all floating-point calculations were scheduled to be executed inside the x87 execution unit. However, as discussed below, all floating-point registers inside the x87 FPU are 80 bits wide (double-extended precision). To mimic calculations done in pure single and double precision (i. e. with 32 and 64 bit wide registers), `IeeeCC754++` explicitly instructs the FPU to use only the same significand widths as in the single and double formats (i. e. 24 and 53 bits, cf. Table 1.1). Note that the width of the exponent cannot be influenced.

This default behaviour (which concerns only the `x87` FPU and the `main` FPU when the compiler maps floating-point instructions to x87 instructions) can explicitly be set via the command line option `--no-extended`; when the option `--extended` is given, all intermediate calculations are done in double-extended precision. `IeeeCC754++` uses `--no-extended` by default.

Finally, the implementation of the `x86 main` FPU is largely identical to that of the `default main` FPU (see Section 5.1.1); however, the "fake" `fma` implementation is missing.

The `x87` FPU

The term x87 has been commonly used for the 8087 FPU (which was initially introduced as a separate coprocessor for the 8086 CPU) and its descendants which were integrated into the CPU on the same die starting with the 80486 processor. For details on the history of the x87 FPU, see [WIK17ak].

For maximum accuracy when executing floating-point operations, the x87 FPU features a so-called double-extended design, implementing an 80 bit FPU with all registers being 80 bits wide. When executing calculations in single or double format, intermediate results are rounded to double-extended precision and only rounded to target precision when writing results back to memory. Although this approach yields better numerical accuracy, it may lead to portability issues, since

results might not be identical when computed on a double-based architecture (for a more detailed discussion, cf. e. g. [MMM13] or [Mul+10, pp. 75 sqq.]).

Note that it is possible to switch the x87 FPU into a state in which operations are not performed in extended precision before rounding to target precision, but correct significand length of the chosen precision is used. This approach avoids double roundings, since only one rounding to correct significand size is needed. However, the size of the exponent is not influenced by this setting. This means that in this state, operations on double precision operands would be performed with a significand length of 53 bits, but with an exponent that is 15 bits wide (instead of the 11 bits used in double precision). This mode of operations was chosen as default since `IeeeCC754++` tries to assess the best possible IEEE-conformity of a user environment, i. e. by default, the `IeeeCC754++ x87` FPU will always use the correct significand size when executing tests. The "extended" precision mode can be selected by using the command line option `--extended`. As a final note concerning precision, the choice of using correct significand length for intermediate calculations has to be kept in mind when executing IEEE-conformity tests since by default (i. e. when a user uses the x87 FPU without enabling this mode), extended precision is used for intermediate calculations.

In order to ensure that floating-point instructions are indeed executed inside the x87 execution unit, the `IeeeCC754++`'s `x87` FPU implements those operations present in the x87 instruction set as inline assembler calls. Amongst these are all basic and conversion operations (cf. Section 1.2.1, page 11) with the exception of conversions between binary and decimal representations. The supported formats are single, double, and extended.

The `3dnow` FPU

3DNow! represents an extension to the x86 instruction set proprietary to AMD and implemented in AMD processors from 1998 to 2010 [AMD00a; AMD00b; WIK17a]. It consisted of a limited number of floating-point instructions that operate on its native data type, a vector of two packed single operands. Only roundTiesToEven is supported.

The `3dnow` FPU in `IeeeCC754++` uses inline assembler implementations of addition, subtraction, multiplication, and conversion to and from integers by directly using the corresponding instructions in the 3DNow! instruction set. For division and square root, no direct equivalents are available, but 3DNow! provides instructions for reciprocal division and square root estimates, as well as instructions for further iteration steps in a Newton-Raphson iteration (see [Sco85; HP11]). Consequently, the `IeeeCC754++` implementation uses an inline assembler version of the corresponding iterations to compute division and square root.

Note that this FPU is implemented only for historical reasons since only computers featuring legacy AMD processors are able to execute instructions from the 3DNow! instruction set.

It should be mentioned that 3DNow! does not represent the first extension to the x86 instruction set aimed at speeding up multimedia computations (such as video and audio processing). In 1997, Intel introduced MMX as a multimedia extension to the Pentium processor. However, since MMX only supports integer instructions, it is not relevant for `IeeeCC754++`.

The `sse`, `ssei`, `sses` FPUs

In 1999, Intel presented the SSE (Streaming SIMD Extensions) instruction set [INT17b, Chapter 10] [WIK17ae], followed by SSE2 in 2000 [INT17b, Chapter 11] [WIK17aa]. SSE introduced a completely new stack for floating-point calculations, including 128 bit wide SIMD registers and a new register for floating-point status. SSE only supports single precision floating-point numbers, which results in the ability to execute four single precision operations with one SIMD call. Support for double precision was added with SSE2.

`IeeeCC754++` implements three FPUs with SSE and SSE2 support, namely `sse`, `sses`, and `ssei`. All three use the instructions supported by SSE: addition, subtraction, multiplication, division, and square root, as well as conversions between floating-point formats and floating-point and integer numbers. While `sse` and `sses` use inline assembler routines for the operator calls, `ssei` implements the operators via compiler intrinsics. The `sse` and `ssei` FPUs make use of the SSE SIMD registers, thereby working on four single or two double values simultaneously. The `sses` FPU implements a scalar version which executes only one floating-point operation on the first entry in the SIMD vector. For details on testing vector units and their scalar versions, cf. Section 3.1.12

Note that this feature (using the SSE units with scalar operands) is also employed by recent compilers for floating-point calculations on x86 platforms: Although the x87 FPU uses larger intermediate precision and therefore yields more precise results, the SSE units offer much higher performance and are chosen by compilers as the default execution unit for floating-point instructions. As an example, `gcc` from version 4.0 on executes floating-point instructions inside the SSE unit if not otherwise directed (and if compiled on a platform that features SSE support).

As a final note, all four classic rounding modes are supported (see above), whereas only single and double precision operands are possible.

The `sse3`, `sse3i`, `sse3s` FPUs

Intel released further instruction set extensions to SSE called SSE3, SSSE3, and SSE4 between 2004 and 2006, adding only a few instructions relevant for floating-point calculations (cf. [INT17b, Chapter 12] and [WIK17ab; WIK17ad; WIK17ac]). `IeeeCC754++` implements only the most relevant functions inside the `sse3`, `sse3i`,

and `sse3s` FPUs:[2] the instruction `ADDSUBPX` which subtracts the values in the first entries of the operands and adds the values in the second entries, `HADDPX` and `HSUBPX` that perform "horizontal" operations on the SIMD vectors,[3] and `ROUNDSX` which rounds values to integral format (`X` standing for either `S` or `D` to denote single or double target precisions). In order to evaluate these instructions, only addition, subtraction, and `rint()` are implemented in the FPUs.

Note that in order to support the `ADDSUBPX`, `HADDPX`, and `HSUBPX` instructions, `IeeeCC754++` features the function `fillVecHorizontal()` that spreads the operands into the SIMD vectors into the appropriate locations (cf. Appendix B.7).

Similar to the SSE case, three variants of the SSE3 FPUs exist: One implementing the operations as inline assembler instructions on SIMD vectors (`sse3`), one using inline assembler on scalars (`sse3s`), and finally an intrinsics vector version (`sse3i`). Note that the `sse` and `ssei` FPUs use `HADDPX` and `HSUBPX` to implement addition and subtraction, whereas the `sses` FPU uses the addition and subtraction parts of the `ADDSUBPX` instruction for the corresponding operation.

The `avx`, `avxi`, `avxsse`, `avxssei` FPUs

Since 2011, another major addition to the x86 instruction set has been supported by processors from both AMD and Intel: AVX (Advanced Vector Instructions) which supports SIMD vectors which are 256 bit wide [INT17b, Chapter 14] [WIK17b]. The supported instructions are mostly AVX variants of all SSE instructions, albeit working on the new larger registers, thereby being able to perform eight single or four double operations with one function call.

AVX2 extended the support for some SSE integer instructions missing in AVX, as well as adding an `fma` instruction fully conforming to IEEE 754-2008. The `avx` FPU once again features an inline assembler implementation targeting the AVX SIMD vectors, while the `avxi` FPU employs compiler intrinsics.

In addition to the "native" mode, the AVX unit in x86 processors can also be used in a legacy SSE mode by using a special instruction prefix (the VEX prefix, see [WIK17ai]) and the first 128 bits of the 256 bit wide AVX registers. The corresponding FPUs inside `IeeeCC754++` are called `avxsse` and `avxssei` (inline assembler and intrinsics versions).

Note that due to compiler limitations, it is not possible to build `IeeeCC754++`'s SSE and AVX FPUs at the same time. Consequently, if the SSE and AVX FPUs should be checked for conformity in the same user environment, two `x86 IeeeCC754++` executables need to be built.

[2] Since only a few floating-point instructions were added to the SSE instruction set with SSE3, SSSE3, and SSE4, they are all collected as "SSE3" FPUs into `IeeeCC754++`.

[3] Given double input vectors $x = (x_0, x_1)$ and $y = (y_0, y_1)$, a regular SIMD addition would yield a result $\texttt{add}(x, y) = x + y =: z = (z_0, z_1)$ with $z_0 = x_0 + y_0$ and $z_1 = x_1 + y_1$. In the horizontal case, the operands for the underlying scalar addition are not taken from both input SIMD vectors, but the values located inside each vector are summed: $\texttt{hadd}(x, y) =: z = (z_0, z_1)$ with $z_0 = x_0 + x_1$ and $z_1 = y_0 + y_1$. For details, see [INT17b].

The avx512, avx512i FPUs

In 2013, Intel proposed 512 bit extensions to the AVX ISA called AVX-512 [INT17b, Chapter 15] [WIK17d]. They consist of a family of extensions supporting vectors of 512 bit length; not all processors implementing AVX-512 must support all extensions. Since the only mandatory extension AVX-512F (AVX-512 Foundation) contains all instructions which are relevant for floating-point computations, `IeeeCC754++` only supports AVX-512F. Currently, AVX-512 is only supported in the Intel Xeon Phi x200 product family [INT17d], also called Knights Landing (KNL), and processors with Skylake microarchitecture [INT17a].

Similar to the SSE and AVX FPUs, `IeeeCC754++` implements `avx512`, an FPU that uses inline assembler instructions, and `avx512i`, a variant employing intrinsics. They are vector FPUs and support all four classic rounding modes.

5.2.2 The mic port

`IeeeCC754++` includes support for KNC and KNL chips, albeit in different form: Since KNL processors are released in a bootable form factor and running the main operating system, the execution model is identical to usual x86 platforms. Therefore, support for KNL is built into `IeeeCC754++`'s `x86` architecture with the `avx512` and `avx512i` FPUs; for details, see Section 5.2.1 above.

For KNC support, the `mic` architecture has been implemented by employing KNC's offload model: `IeeeCC754++` itself is run on the host CPU, and only the actual floating-point operations are offloaded to the KNC accelerator device via compiler pragmas. The `main` FPU is implemented in a manner similar to the `x86` `main` FPU, but the actual floating-point operations are offloaded to the KNC.

For all three FPUs, an `IeeeCC754++` executable built for the `mic` architecture supports two additional command line arguments: `-scan` scans for KNC devices attached to the host system and prints the number of devices found, and `-device=<DEVICE>` selects the attached KNC device number `<DEVICE>` for the execution of the offloaded floating-point operations. If n devices are found on the platform, a value in the range of $0, \ldots, n-1$ selects the respective device whereas a value of -1 leaves the choice to `IeeeCC754++` (or rather, the software library responsible for initialisation and setup of the KNC devices).

The avx, avxi FPUs

The `mic` port contains two variants of FPU that make use of the SIMD units built into KNC processors: `avx` which has been implemented with inline assembler instructions, and `avxi` using intrinsics.

5.3 The ARM architecture

"ARM [...] is a family of reduced instruction set computing (RISC) architectures for computer processors, configured for various environments." [WIK17c] Processors based on the ARM architecture still play a comparatively minor role in HPC, despite its dominant role in the mobile market where almost all smartphones and most tablets feature CPUs based on some variant of the ARM ISA. The architecture is developed by the company ARM Holdings which itself does not manufacture semiconductor chips. Instead, it develops the ISA and hardware designs for processor cores based on that ISA and licenses the IPs of these components according to the following models [Shi13]:

- A *processor license* is the license to use a CPU design from ARM Holdings. How the design is implemented into silicon is up to the licensee, as long as the design itself is not changed. ARM Holdings provides general guidelines as to how these designs should be implemented, but the process can be adapted as needed.

- With a *processor optimisation pack (POP)*, ARM Holdings licenses an optimised processor design manufacturable at a specific semiconductor foundry. POPs are available for various processor/foundry/process node combinations. This model denotes the easiest way to produce a CPU with ARM ISA, but it does not allow for application specific optimisation e. g. with regard to maximum performance or increased power efficiency.

- An *architecture license* allows for the custom implementation of the licensed ARM ISA into the customer's product. It can be freely designed and implemented into the resulting processor as long as it follows the respective ISA.

Due to these licensing models, a huge variety of slightly different processors based on ARM ISA exists. Furthermore, different ISAs have been developed over the last decades, mainly aimed at scenarios where low power consumption is desired, e. g. embedded devices such as control boxes modules in manufacturing or automotive applications, or smartphones or tablets for which extended device runtime with one battery charge is desirable. According to [WIK17c], "ARM is the most widely used instruction set architecture in terms of quantity produced."

Starting in the 1980s, different variants of the ARM architecture have been developed as 32 bit architectures, the currently most widespread one being ARMv7-A. In 2011, ARM Holdings announced ARMv8-A which introduced a new 64 bit ISA, sometimes also called AARCH64. `IeeeCC754++` supports both variants, but due to differences in the instruction sets in separate architectures: The `arm` architecture supports all 32 bit variants, whereas the `aarch64` architecture supports ARMv8-A in 64 bit mode.

5.3.1 The `arm` port

With regard to floating-point computations in the `arm` architecture, two extensions to the ARM ISA are most relevant: VFP and NEON (see below). In order to check ARM processors without these extensions for IEEE-conformity, the `main` FPU can be used which is implemented in a manner almost identical to the `main` FPU in the `default` architecture.

The `vfp`, `vfps`, `vfpv4`, `vfpv4s` FPUs

VFP (vector floating point) is a coprocessor floating-point extension to ARM instruction sets up to ARMv7-A. It provides instructions for the basic arithmetic operations including square root and `fma` for single and double precision floating-point numbers, as well as corresponding VFP floating-point registers. Note that the "vector" acronym is slightly misleading, since VFP operates on vectors of size up to 256 bits, albeit not in true SIMD fashion: The operations on vector elements are not executed in parallel, but one after the other.

`IeeeCC754++` implements two FPUs which support two variants of VFP: `vfp` which can be built for VFP up to version 3, and `vfpv4` which supports version 4 and includes support for actually fused `fma` (i. e. multiply and addition without intermediate rounding – earlier VFP version include a multiply-add instruction that is not fused). Furthermore, `IeeeCC754++` provides scalar variants of both FPUs named `vfps` and `vfpv4s` that do not operate on VFP vectors, but instead on single VFP values stored in VFP registers, and that add support for conversions between floating-point numbers and conversions to and from 32 bit integer values. All VFP instructions are implemented as inline assembler calls, including setting of the requested rounding mode and retrieval of floating-point exceptions. Finally, one caveat regarding vector support in newer ARM needs to be mentioned: ARMv7-A deprecates use of vectors in the VFP FPUs, i. e. VFP should only be used with scalar operands (cf. [ARM14]).

The `neon`, `neonq`, `neoni`, `neonqi` FPUs

In ARMv7-A, NEON denotes an extension to ARM instruction sets geared towards speeding up multimedia applications. In contrast to VFP, it is not specifically tailored for floating-point computations, resulting in a much broader instruction set including integer and fixed-point instructions. Floating-point instructions are included only for single precision. NEON is usually implemented in true SIMD fashion, i. e. it supports parallel execution. NEON and VFP need not both be implemented at the same time; however, when both extensions are present, they share the same set of processor registers. The NEON unit can view a register bank as either sixteen 128-bit quadword registers called Q0-Q15 or thirty-two 64-bit doubleword registers called D0-D31.

`IeeeCC754++` implements the following operations contained in the NEON instruction set: the basic arithmetic operations including square root and conversion to and from 32 bit integers. Variants using inline assembler instructions and intrinsics exists, as well as variants working on D type registers (64 bit, i. e. vectors of two singles) and Q type registers (128 bit, i. e. vectors of four singles). The four resulting FPUs are then called `neon`, `neonq`, `neoni`, and `neonqi`, with `q` indicating Q type registers and `i` denoting the intrinsics version.

5.3.2 The `aarch64` port

The `aarch64` architecture denotes `IeeeCC754++`'s implementation of the 64 bit ARMv8-A instruction set (cf. Section 5.3). With ARMv8-A, VFP is no longer supported, and NEON has been renamed to Advanced SIMD and significantly expanded: ASIMD now includes full double precision and 64 bit integer support, and the number of registers has been doubled. Furthermore, in contrast to ARMv7-A, the ASIMD/NEON unit is a mandatory part of the ARMv8-A ISA. In addition to these FPUs, ARM Holdings recently introduced a new vector unit called SVE aimed primarily at HPC applications (for details, see below).

`IeeeCC754++` supports all these FPUs as well as a general implementation of the `main` FPU (which is identical to the `arm` architecture `main` FPU).

The `neoni`, `neonqi`, `asimd` FPUs

`IeeeCC754++` implements NEON/ASIMD support by using inline assembler and intrinsics versions. All FPUs support the basic arithmetic operations including square root and `fma`, as well as the full set of conversions (excluding conversion between binary and decimal formats).

The inline assembler version is called `asimd` to follow the new naming. However, since compilers still use the NEON intrinsics naming scheme for backwards compatibility with programs written for ARMv7-A, the ASIMD FPUs in `IeeeCC754++` using intrinsics to execute floating-point operations reflect this naming and are called `neoni` and `neonqi`, with `i` denoting the use of intrinsics and `q` indicating that Q type registers have been used (cf. Section 5.3.1). In contrast to the ARMv7-A version of these FPUs, operands are supported in single and double precision. As a consequence, the `neoni` FPU operates on vectors of either two singles or one double (thereby not being a "real" vector operation), whereas Q type vectors in `neonqi` comprise four singles or two doubles. Vectors in the `asimd` FPU are always 128 bit wide.

The `sve` FPU

In 2016, ARM Holdings announced SVE (Scalable Vector Extension), a SIMD vector unit targeted at HPC applications. It operates on vectors ranging from 128 to 2048 bit length and claims full IEEE 754-2008 support for floating-point

operations. The choice of actual vector width inside the processor hardware is left to the manufacturer (as long as the vector size is a multiple of 128 bits). However, in order to avoid recompiling applications for processors with SVE implementations with differing vector sizes, the ARM ISA supports a VLA (vector length agnostic) programming model enabling users to write programs for SVE which can be executed on every platform that supports SVE, regardless of actual (hardware) vector length.

As of the writing of this thesis, no hardware implementations of SVE exist, although SVE enabled versions of `clang` and `gcc` are already available. The `sve` FPU implemented inside `IeeeCC754++` has been developed and tested for functionality with the ARM instruction emulator `armie` [ARMa]. In order to accommodate for the VLA programming model, `IeeeCC754++` always operates on vectors that are 2048 bit wide. As a consequence, while hypothetical hardware using 2048 bit vectors would execute a floating-point operation on all vector elements inside the `sve` FPU with only one instruction, another processor with (again hypothetical) 512 bit vector length might need four instructions for the same result.

The `sve` FPU implements all basic arithmetic operations including square root and `fma` for single and double operands, as well as all conversions between floating-point and integer formats. Note however that when converting a floating-point value to an integer, rounding support is only available for roundTowardZero.

5.4 The Power Architecture

Processors and computers based on some variant of the Power Architecture have a long and ramified history, making identification of the ISA of a given "Power" processor non-trivial due to various instantiations of instruction sets and architectures, various similar names for these, and features of different versions of the respective ISAs being inherited and shared by only some ISAs. The POWER history starts in 1990 when IBM introduced their first computer based on the "POWER architecture" denoting a RISC ISA developed for mid-range workstations and servers. A few years later, an alliance of Apple, IBM, and Motorola developed a mass-market version of the POWER processor based on a so-called "PowerPC architecture". Since then, different variants of processors based on similar but differing POWER ISAs have been released by different manufacturers, with IBM being the most prominent developer. In 2004, IBM and 15 other companies founded Power.org [WIK17w] "as an organisation with the mission of developing products revolving around the Power Architecture and with the purpose of developing, enabling and promoting Power Architecture technology" [WIK17v]. Processors based on some variant of POWER ISA have been deployed in a broad range of computing devices from desktop gaming consoles (e. g. Cell in the Sony PS 3, Xenon in the Microsoft XBOX 360, and Broadway in the Nintendo Wii), desktops

and notebooks (especially in Apple products such as iBook, PowerBook, iMac, Mac Mini, etc.), servers (IBM System i, System p, and Power Systems), and HPC systems such as IBM's Blue Gene line or QPACE which was based on the Cell processor. In order to gain better access into the x86 dominated server market, IBM founded the OpenPOWER Foundation in 2013, which opens up the POWER8 (and POWER9) ISA for licensing to different manufacturers. Table 5.2 shows an overview of the different names and ISAs related to the Power Architecture. For more details on the history and variants of the POWER ISA, cf. [WIK17v].

Term	Description
POWER	Performance Optimization With Enhanced RISC. An old microprocessor instruction set architecture designed by IBM.
PowerPC	Power Performance Computing. A 32/64-bit instruction set for microprocessors derived from the POWER ISA, including some new elements. Designed by Apple, IBM and Motorola.
PowerPC-AS	PowerPC-Advanced Series. Codename "Amazon". A purely 64-bit variant of PowerPC, including some elements from the POWER2 version of the POWER ISA. Used in IBM's RS64 family processors and newer POWER processors.
POWERn	Where n is a number from 1 to 9. A series of high-end microprocessors built by IBM using different combinations of POWER, PowerPC, PowerPC-AS and Power ISAs.
Cell	Cell Broadband Engine Architecture (CBEA), a microprocessor architecture designed by IBM, Sony and Toshiba, which has Power Architecture as a part.
Power Architecture	The broad term designating all that is POWER, PowerPC and Cell including software, toolchain and end-user appliances.
Power ISA	A new instruction set, combining late versions of POWER and PowerPC instruction sets. Designed by IBM and Freescale.

Table 5.2: *Power Architecture: Names and ISAs. Mostly taken from [WIK17v].*

Originally developed as a 32 bit big-endian RISC ISA, IBM introduced the first 64 bit variant with POWER2 in 1995. Current Power Architecture processors are bi-endian, i. e. they support loading and storing both big-endian and little-endian data. With the release of POWER8 in 2014, IBM chose little-endian as the default access type. For more details, see [WIK17j]. Most POWER processors support single and double floating-point numbers in hardware, while extended precision is available in software with the IEEE 754-2008 quadruple format, i. e. the `C` data type **`long double`** maps to an 128 bit quadruple format (in contrast to x86 where **`long double`** usually maps to Intel's 80 bit double extended format).

`IeeeCC754++` tries to support the different variants of POWER ISA by supplying a "generic" POWER architecture port called `ppc`[4] which has been successfully

[4]The acronyms `ppc` for PowerPC and `ppc64` for the respective 64 bit variant are used by the Linux operating system. The naming in `IeeeCC754++` reflects this choice.

tested on Cell, POWER7, and POWER8 processors. It includes support for different SIMD units, namely `altivec` and `vsx`, as well as `ppu` FPUs mainly relevant for the Cell processor, but also able to be used on POWER7 with corresponding libraries. The `ppc` architecture features implementations of the conversion operators between `IeeeCC754++`'s internal floating-point format and the machine formats for both big- and little-endian configurations (although the big-endian mode is only needed for POWER8 currently).

In addition to the `ppc` architecture, `IeeeCC754++` implements two architecture ports aimed at POWER architectures, differing to such a degree that supporting these architectures inside the `ppc` architecture is technically not viable: Blue Gene/Q, which denotes an architecture targeted at large-scale HPC applications including a specialised SIMD unit called QPX (see Section 5.4.3 for the `bgq` architecture), and the Cell processor which was developed by a consortium of Sony, Toshiba, and IBM for a broad range of applications, cf. Section 5.4.2 which describes the `cell` architecture.

5.4.1 The `ppc` port

The `main` FPU in the `ppc` architecture supports single, double, and quadruple formats in little- and big-endian formats. All operators implemented in the `default main` FPU are also supported in the `ppc main` FPU, with the exception of `fma`. Since the other `ppc` FPUs add support for different hardware floating-point extensions, they do not implement quadruple precision.

The `altivec`, `vsx` FPUs

AltiVec is a single precision floating-point and integer SIMD instruction set designed and owned by Apple, IBM, and Freescale (formerly Motorola) which was developed between 1996 and 1998. Since AltiVec is a trademark owned by Freescale, it is also referred to as Velocity Engine by Apple and VMX (Vector Multimedia Extension) by IBM. It provides 32 registers of 128 bit length and instructions for the most common floating-point operations, including `fma`. Note that there is no direct support for division and square root. Instead, instructions for estimates of reciprocals and reciprocal square roots are provided. `IeeeCC754++` emulates division and square root by using these instructions together with a Newton-Raphson iterative approach, cf. Section 5.2.1.

In 2010, IBM introduced a SIMD unit called VSX (Vector Scalar Extension) which extends AltiVec/VMX to support up to 64 floating-point registers, operations on doubles, and instructions for floating-point division and square root. It was first implemented in POWER7 processors.

`IeeeCC754++` implements support for AltiVec/VMX inside the `altivec` FPU which allows only for single operands and uses Newton-Raphson for division and square root. Only conversion instructions from and to 32 bit integers are supplied

(by AltiVec and consequently the implementation inside `IeeeCC754++`), as well as rounding a floating-point number to an integral value. The implementation is done via intrinsics; all four classic rounding modes are supported.

The `vsx` FPU is also implemented using compiler intrinsics. However, due to `gcc` and IBM `XLC` using different mnemonics for load and store operations, two versions exist (which are automatically chosen by the `IeeeCC754++` build system depending on the chosen compiler): `vsxgcc` and `vsxxlc`. The same operations and rounding modes as in the `altivec` FPU are supported; however, the `vsxxlc` variant additionally supports conversion between single and double floating-point formats and converting to 64 bit integers. Note that for `gcc` 4.3, only intrinsics for single precision operations are available.

The `ppu` FPU

The PPU (Power Processing Unit) is the main processing unit used in the Cell processor. A more thorough description of the PPU design (and the Cell design in general) is given in Section 5.4.2. The `ppu` FPU (and the corresponding SIMD variants `ppusimd` and `ppusimdi`, see below) are implemented as FPUs for the `ppc` Architecture for the following reasons: Since it represents an implementation of a 64 bit Power Architecture, some of its features, especially concerning floating-point execution, are available in other POWER ISAs, and IBM makes the corresponding intrinsics available via the `C` header file `ppu_intrinsics.h`. Consequently, these FPUs are not implemented as Cell FPUs, but for the more generic `ppc` architecture, although not all of these PPU instructions are necessarily available on every POWER platform (such as a (scalar) square root implementation). The `ppu` FPU has been confirmed to work on Cell and POWER7 CPUs.

The `ppu` FPU is a scalar FPU supporting the following floating-point operations for single and double operands: multiplication, square root, `fma`, and conversion from and to 64 bit integers. Furthermore, `IeeeCC754++` implements addition and subtraction by using fused multiply-add (and multiply-subtract[5]), since there are no regular addition and subtraction operators. Division is once more supported via a reciprocal estimate operation and a following Newton-Raphson iteration. All classic rounding modes are supported.

The `ppusimd`, `ppusimdi` FPUs

In the Cell CPU, the main floating-point compute capabilities and performance is provided by up to eight SPUs (see Section 5.4.2). These are usually accessed using a specifically tailored set of function calls, collected in a SIMD library called `spusimd`. In order to provide for an (almost) identical way of accessing the

[5]Fused multiply-subtract is a variant of fused multiply-add where a multiplication is followed by a subtraction instead of an addition. Note that this "`fms`" operator can make use of `fma` hardware since only the sign bit of the third operand needs to be flipped.

floating-point capabilities in the PPU, IBM provides identically named functions inside the SIMD library `ppusimd`. These instructions target the VMX FPU inside the PPU (which supports only single precision operands), and the following operations are supported for all rounding modes (implemented in the `ppusimd` FPU in `IeeeCC754++`): division, remainder, square root, `fma`, round to integral value, and conversion to (32 bit) integers.

There are two ways to make the instructions available inside a `C` program: by either importing all function definitions via the header file `simdmath.h`, or by importing only specific functions via single header files such as `simdmath/fmaf4.h` to use the `_fmaf4` intrinsic that executes an `fma` operation on a vector of four single floating-point numbers. `IeeeCC754++` supplies two FPUs to reflect these different ways of accessing the floating-point instructions: The `ppusimd` FPU imports all function definitions by using the header file `simdmath.h`, and the `ppusimdi` FPU uses the second variant, i. e. it includes header files for every supported floating-point instruction. Apart from this, the FPUs are identical.

5.4.2 The `cell` port

"Cell is a multi-core microprocessor microarchitecture that combines a general-purpose Power Architecture core of modest performance with streamlined co-processing elements which greatly accelerate multimedia and vector processing applications, as well as many other forms of dedicated computation. [...]. It was developed by Sony, Toshiba, and IBM [...]. Cell is shorthand for Cell Broadband Engine Architecture, commonly abbreviated CBEA in full or Cell BE in part." [WIK17f] The most notable deployments of the Cell processor (or its newer and slightly modified successor PowerXCell 8i) are in the Sony PS 3 gaming console and the IBM Roadrunner [WIK17p] and QPACE [Bai+09; WIK17x] supercomputers.

The Cell consists of an PPU (see Section 5.4.1) which serves as main CPU and up to eight coprocessing elements called SPU (Synergistic Processing Unit) which provide computing performance for multimedia and scientific workloads. The PPU includes a VMX unit that supports single precision SIMD floating-point operations on 128 bit wide vectors (cf. Section 5.4.1), but due to its limited performance, it is rarely used for floating-point computations. On the other hand, each SPU consists of a SIMD unit working on vectors that are 128 bit wide, and a moderately sized but very fast memory called local store. It allows for SIMD floating-point operations on either single or double precision operands, with the double precision operations being up to eight times slower compared to operations on single precision operands in the earlier Cell versions, and only taking twice the time on the PowerXCell 8i.

The floating-point facilities of the SPU are not fully IEEE-conforming. For single precision, not being conforming was a design decision to allow for fast and consistent execution times of the floating-point operations. Therefore, only the roundTowardZero rounding mode is supported (implemented by simply truncating

the result, i. e. by dropping all bits after the last place of the significand), and all subnormal numbers are flushed to zero (FTZ and DAZ). Additionally, no NaNs are supported. Instead, the (unbiased) exponent $e_x = U + 1 = 255$ is used as a regular exponent, allowing for larger normalised floating-point numbers than specified by IEEE 754-2008. For double precision, the implementation is almost IEEE-conforming: All four classic rounding modes are supported, and the maximal exponent is used for infinity and NaN representation. However, FTZ and DAZ are also used for double precision operands and results. For details on the implementation of the floating-point capabilities of the SPU, we refer to [Mue+05].

Since the PPU is based on POWER ISA, its normal floating-point capabilities including the VMX unit can be tested via the `ppc` architecture and its `main` and `altivec` FPUs. Furthermore, the special capabilities built into the PPU can be tested by using the `ppu`, `ppusimd`, and `ppusimdi` FPUs (cf. Section 5.4.1). As a consequence, `IeeeCC754++` implements only SPU support with the `spu`, `spusimd`, and `spusimdi` FPUs inside the `cell` architecture and does not make use of the `main` FPU (i. e. is does not implement any operations).

Note that to compile `IeeeCC754++` with the `cell` architecture and either of the three FPUs, the Cell SDK version 3.1 is needed. The `cell` architecture inside `IeeeCC754++` contains the code necessary to create and spawn threads on an SPU, as well as code that handles data transfer from main memory to local store and executes the desired floating-point operation on a SIMD vector inside an SPU. The resulting executable provides two additional command line parameters: By default, every test vector is executed simultaneously on all SPUs (and the results are evaluated) before proceeding with the next test vector. `--spus=<SPUs>` limits the number of SPUs used for the floating-point execution to `<SPUs>` SPUs. In particular, with `--spus=1`, only the first SPU is employed for IEEE-conformity testing. Additionally, the parameter `--nosched` binds the SPU contexts created by `IeeeCC754++` to the respective SPUs on which they have been created and prevents the operating system from scheduling a context on a different SPU. This approach ensures that indeed all execution units on all SPUs are tested by spawning one context on every SPU and using that context for testing.

The `spu` FPU

The `spu` FPU employs the intrinsics supplied by the Cell SDK to target the floating-point operations in the SPU instruction set as directly as possible. As a consequence, the list of supported operations and precisions reflects the choices of the SPU ISA designers: $+$, $-$, $*$ and `fma` are supported for single and double operands. There are no operations that directly supply division and square root, but only reciprocal and reciprocal square root estimates, and those are only supplied in single precision. The `spu` FPU therefore implements single precision division and square root via the Newton-Raphson iterative approach. Also, conversion between floating-point and integer formats is only supported

between singles and 32 bit integers. Finally, converting from double to single precision floating-point numbers is contained in the instruction set.

Due to Cell being a discontinued architecture that is not widely deployed anymore, `IeeeCC754++` does not implement handling of rounding modes or exceptions on the SPUs. Therefore, the `spu` FPU only supports roundTowardZero for single precision (due to SPU limitations, see above) and roundTiesToEven (the default rounding mode) for double precision operators. It should be kept in mind that the SPU uses FTZ and DAZ, i. e. for meaningful testing results, the command line option `--nosubnormal` should be used.

The `spusimd`, `spusimdi` FPUs

Since the intrinsics provided for the SPU are incomplete from an IEEE-conformity point of view, the Cell SDK provides an (almost) complementary SIMD library called `simdmath` that supports the following operations: division, remainder, square root, and `fma`, as well as rounding to integral value and converting floating-point numbers to 64 bit integers. All of these are available for single and double operands for all four classic rounding modes.

The `simdmath` library can be used for computations on the PPU and the SPU; depending on the context, either VMX instructions (or software equivalents) or SPU SIMD instructions are generated by the compiler. The functions are provided in one of two ways: The whole `simdmath` library can be imported into a `C` program via the header file `simdmath.h`, and it is also possible to only include specific functions by including their prototype via specific header files such as e. g. `simdmath/sqrtf4.h` and `simdmath/sqrtd2.h` for single and double square roots. The `spusimdi` FPU uses the latter approach while the `spusimd` FPU includes all functions at once. The naming is thus identical to that of the `ppusimdi` and `ppusimd` FPUs in the `ppc` architecture (cf. Section 5.4.1).

5.4.3 The `bgq` port

IBM's Blue Gene [IBM11b; WIK17e] is a line of supercomputers designed for the HPC space. Numerous installations of the three models that have been available since 2004 (Blue Gene/L, Blue Gene/P, and Blue Gene/Q) topped the Top500 [TOP500b], Green500 [Green500], and Graph500 [Graph500] lists, with Blue Gene/Q installations leading all three lists at the same time in June 2016 [WIK17e]. All Blue Gene machines are massively parallel multiprocessing computers with custom highspeed networks, enabling the parallel execution of jobs with up to hundreds of thousands of cores.

The latest incarnation Blue Gene/Q [IBM11a] employs processing cores called IBM A2 that are based on POWER ISA. Each processor contains 18 computing cores, 16 of which are used for actual computations, one is reserved for the operating system and input/output, and one is designated as spare should one

of the other cores fail. Each core contains four double precision pipelines with 256 bit wide vector registers usable for scalar operations or SIMD operations on four double precision vectors.

`IeeeCC754++` implements the `main` FPU in the `bgq` architecture as a big-endian version of the `default main` FPU. This means it supplies support for all operations and rounding modes on single, double, and quadruple floating-point numbers.

An additional note is needed for Blue Gene/Q's execution model: Since Blue Gene/Q is targeted at HPC applications, the system design allows for a smallest subset of one node (which consists of 32 A2 processors) to be allocated as compute resources. In order to run `IeeeCC754++` (which is a serial program), the `bgq` architecture must be compiled with MPI support. Upon start, `IeeeCC754++` determines which processor (rank in MPI terms) it is running on and terminates itself on all but the first processor.

The `qpx`, `scalar` FPUs

In `IeeeCC754++`, Blue Gene/Q's SIMD unit is supported by two FPUs which use different sets of intrinsics: The `scalar` FPU employs intrinsics that perform scalar floating-point operations on single and double operands via an interface similar to the PPU intrinsics used in the `ppc ppu` FPU (cf. Section 5.4.1). The arithmetic operations $+$, $-$, $*$, square root, and `fma` are supported, as well as rounding to integral value and converting from floating-point numbers to integers. Furthermore, instructions for division exist; as their names `__swdiv` and `__swdivs` imply, the actual division routines do not call equivalents implemented in hardware, but rather emulate the operation in software. All four classic rounding modes are supported.

The SIMD unit implemented in the Blue Gene/Q processor is called QPX (Quad Processing eXtension to the Power ISA)[Fox12]. Every QPX vector is 256 bits wide and contains four double precision floating-point numbers. Consequently, only instructions performing floating-point operations on doubles are supplied. The `IeeeCC754++ qpx` FPU makes use of these instructions when testing double precision operations; single precision operations are implemented inside the `qpx` FPU by converting the operands from single to double, performing the QPX operation, and converting the resulting double value back into single format. Obviously, this double rounding (the first rounding occurring during the floating-point operation and the second when converting to single) might lead to wrong results. Therefore, this special setup for the `qpx` FPU has to be kept in mind when evaluating the IEEE-conformity of QPX.

QPX supports all four classic rounding modes, and the following floating-point operations are supplied in hardware (and implemented inside `IeeeCC754++`): $+$, $-$, $*$, `fma`, rounding to integral value, and converting double into single values. Furthermore, software instructions are provided for division and square root. In this manner, it is possible to use division and square root inside algorithms which

are implemented for QPX, although the execution times for these operations are higher than the hardware implementations for e. g. addition or `fma`. Finally, since only double values are allowed in QPX vectors, the QPX instruction set contains no routines to convert between floating-point numbers and integers.

5.5 GPUs and accelerators

In this section, we discuss accelerators in the sense of hardware that complements the main processor built into a computing environment by supplying additional computing power, typically targeted at specific computing needs (see also Section 1.4.2, page 28). In this thesis, we consider only two types of accelerators: GPUs (see below) and the Intel KNC chips (cf. Section 5.2.2). Note that support for other accelerators such as FPGAs can be added to `IeeeCC754++` by implementing a custom architecture port, see Section 3.1.5 and Appendix B.

Originally, GPUs (graphics processing units) were developed as extension cards dedicated exclusively to generating video output for computer monitors, e. g. to support graphically demanding computer games or CAD (computer aided design) programs where using a dedicated GPU card would result in tremendously faster video output. Around 2001, GPUs gained support for programmable shaders and floating-point numbers, originally to enable more complex graphics calculations [WIK17n]. Since GPUs offered an excellent price versus (theoretical peak) performance ratio, scientists soon began to port numerical algorithms to GPUs. In order to achieve this, toolkits and APIs such as OpenGL and DirectX had to be used targeted at and more suitable for actual graphics processing on GPUs. General purpose computing on GPUs (also called GPGPU) reached mainstream status when NVidia released the first version of CUDA (Compute Unified Device Architecture) in 2007. The importance of computing on GPGPUs is illustrated by the publication of books such as [SK11] or [CS15].

Technically, GPGPUs can be seen as accelerators which execute a large number of numerical calculations concurrently. Memory and computing units in GPUs are organised in blocks that contain a large number of identical execution units. Typically, each block can execute a certain number of computations in a manner similar to the SIMD programming model used in vector execution units. All current GPUs and toolkits support floating-point computations and claim to be IEEE-conforming. Usually, floating-point operations on half, single, and double operands are implemented. The dominant frameworks are CUDA for NVidia GPUs and OpenCL that supports GPUs from all major vendors (as well as computations on the CPU through the same API).

The execution model for algorithms on GPUs is significantly different from the CPU execution model: Algorithms are implemented as compute kernels which are then compiled into code that is executable on the GPU. The execution of the kernels is triggered by the CPU through the API used (e. g. CUDA or OpenCL).

The data on which the computation operates must either be already available on the GPU or explicitly transferred to and from the device. The smallest execution unit is one block of shaders (i. e. numerical execution units). Each block can run a certain number of threads executing the same kernel concurrently on the desired data. From a programming point of view, the execution of a kernel can be seen as a SIMD operation on numerical data.

`IeeeCC754++` adopts this execution model: The floating-point operations to be tested are implemented as GPU compute kernels and executed as threads inside blocks. To achieve this, the number of blocks and the number of threads per block can be explicitly given via the commandline parameters `--blocks=<BLOCKS>` and `--threads=<THREADS>` (with defaults of 16 blocks and 16 threads per block). The resulting vector length l_v is then given by $l_v =$ `<BLOCKS>` $\cdot$ `<THREADS>`, with a default of $l_v = 256$. `IeeeCC754++` initialises the respective input vectors with the operands, transfers the data to the GPU, executes the requested number of threads, and retrieves the result vector. Afterwards, the result is evaluated as usual.

Two additional parameters are important when executing `IeeeCC754++` on GPUs: `--scan` tries to detect the respective accelerators (GPUs in the case of CUDA and GPUs and CPUs in the case of OpenCL) and lists the available devices. The complementing parameter `--device=<DEVICE>` triggers execution of the floating-point operations on the selected device, with `<DEVICE>` being the device number as reported by `--scan`.

5.5.1 The `nv` port

The `nv` architecture in `IeeeCC754++` targets NVidia GPUs via CUDA, which "is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels" [WIK17h]. Since the first introduction of CUDA, GPUs have improved significantly, resulting in more and advanced functionality available. CUDA handles the differing feature sets resulting from different versions of the CUDA toolkit, as well as different GPU cards, by collecting features in so called "compute capabilities". A backward-compatible approach is used, i. e. programs compiled with support for lower versions of compute capability can be run on newer hardware supporting higher versions of compute capability. From a floating-point point of view, operations on single precision operands have been supported from compute capability 1.0. The most notable changes with newer versions are double support starting with compute capability 1.3, rounded versions of division and square root which are fully IEEE-conforming from compute capability 2.0, and finally half precision support implemented in compute capability 5.3 and newer (cf. [NVi17a]).

It is important to note that code compiled for compute capability versions higher than available on a specific GPU will result in incorrect numerical output – in tests, all vectors returned by such an executable were always zero. Therefore, it is

advisable to compile `IeeeCC754++` targeting a certain GPU with the exact compute capability version supported by that GPU by using the `configure` parameter `--with-sm=<CC>` where `<CC>` is the supported compute capability version. The default for `<CC>` when `configure` is called without a `--with-sm` parameter is 2.0, thereby enabling full double support.

The `cuda`, `cuda_rn` FPUs

CUDA supports the usual floating-point operations such as the basic operations $+$, $-$, $*$, $/$, remainder, square root, `fma`, and all conversions except between binary and decimal formats (which are not relevant on an accelerator). In addition, all elementary functions defined by `C99` are implemented. However, no floating-point exceptions are supported. Since compute capability 2.0, all operations support IEEE 754-2008 (except exceptions) and in particular subnormals – in older versions, some operations such as division and square root were lacking subnormals support, resulting in subnormal results being flushed to zero.

The `main` FPU adopts an implementation style similar to the `default main` FPU: The basic arithmetic functions are executed by using the regular `C` operators, and conversions are supported by functions with the same names as the corresponding `C` functions. Floating-point operations are executed on scalar operands, not on vectors. Note that setting rounding modes by modifying the environment is not supported. Therefore, all operations are executed in roundTiesToEven mode. `IeeeCC754++` implements support for single and double precision operations for all FPUs inside the `nv` architecture.

The `cuda` FPU is basically identical to the `main` FPU, but executes floating-point operations on vectors in a SIMD fashion, launching the requested number of blocks and threads on the GPU. Finally, the `cuda_rn` FPU accommodates for the fact that only roundTiesToEven is supported: It uses the regular `cuda` FPU implementation by directly calling the operators contained in latter FPU, but it registers only the roundTiesToEven rounding mode.

The `cudai` FPU

In addition to the CUDA `C`-style operators which only support roundTiesToEven, the CUDA API provides intrinsics versions of all operators except floating-point remainder. These intrinsics exist in four versions, each supporting one of the four rounding modes roundTiesToEven, roundTowardPositive, roundTowardNegative, and roundTowardZero. The `cudai` FPU inside `IeeeCC754++` makes use of these intrinsics to implement a fully rounded version of CUDA operands. Note that since exceptions support is not implemented on GPUs, it also not available for these functions.

5.5.2 The `opencl` port

While NVidia's CUDA supports only GPUs (and only those built by NVidia itself), the OpenCL (Open Computing Language) API targets a much broader spectrum of computing devices: "OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs), and other processors or hardware accelerators. OpenCL specifies programming languages (based on C99 and C++11) for programming these devices and application programming interfaces (APIs) to control the platform and execute programs on the compute devices." [WIK17u]

OpenCL views a platform as consisting of a number of compute devices, which might be CPUs or accelerators such as GPUs, attached to a host processor (usually the CPU). Similar to CUDA, functions are executed as compute kernels and must be compiled for the target device prior to execution. The smallest computing unit in an OpenCL device is called PE (processing element), with a compute unit consisting of several PEs. In practice, OpenCL is mainly used to write portable code for GPGPUs, since OpenCL denotes an open standard adopted by all major CPU and GPU vendors (in contrast to NVidia's proprietary CUDA API). In order to describe the execution units in GPUs that actually perform floating-point operations, `IeeeCC754++` uses the more general notion of blocks and threads even in the `opencl` architecture.

In order to provide a uniform interface to GPUs regardless of the API used (either CUDA or OpenCL), `IeeeCC754++` implements a set of commandline parameters in the `opencl` architecture almost identical to the `nv` architecture, i. e. the parameters `--threads`, `--blocks`, `--scan`, and `--device` are supported to select block size and number of threads and to detect which devices are present on the current platform. Additionally, due to OpenCL targeting a broader range of devices, two additional parameters are supported: `--platform=<PLATFORM>` selects a platform as detected by `--scan` (the available choices usually consisting of the host CPU and an accelerator such as a GPU), and `--type=cpu|gpu` selects either the first CPU or the first GPU found on the current platform as target device for the execution of floating-point operations.

The `opencl`, `opencl_rn` FPUs

Similar to `IeeeCC754++`'s `nv` architecture (see Section 5.5.1), the `main` FPU inside the `opencl` architecture implements scalar operations on the target device via the OpenCL API. It calls the usual `C` operators, as well as conversion routines named after the corresponding `C` functions. Setting rounding modes or retrieving floating-point exceptions is not supported; all operations are executed in roundTiesToEven mode. The `main` FPU nonetheless registers all four classic rounding modes. The reason for this decision lies in four operators being available that perform rounding a floating-point number to an integral value in different manners, namely

`opencl_rint`, `opencl_ceil`, `opencl_floor`, and `opencl_trunc`. Therefore, the round to integral operator has been implemented in `IeeeCC754++` by making use of these functions. `IeeeCC754++` supports single and double precision operands in all FPUs in the `opencl` architecture.

Mimicking the implementation of FPUs in the `nv` architecture, two additional almost similar FPUs exists for OpenCL: The `opencl` FPU uses the same operators as the `main` FPU, but works on SIMD vectors (whose length is determined by the requested numbers of blocks and threads, see above), and the `opencl_rn` FPU uses all operators from the `opencl` FPU, but registers only the roundTiesToEven mode.

The `opencl_round` FPU

OpenCL only provides versions of the arithmetic functions using roundTiesToEven. However, for conversions between floating-point formats and between floating-point and integer formats, directed rounded versions exist. Similar to the intrinsics with rounding support in the `nv cudai` FPU, every conversion routine exists in four versions, each implementing one of the four classic directed rounding modes. `IeeeCC754++` implements rounded versions of these conversions inside the `opencl_round` FPU by selecting the corresponding OpenCL function depending on the currently tested rounding mode. Once again, floating-point exceptions are not supported.

5.6 In-network computations

In Section 1.4.8, we discuss the advent of in-network floating-point computations which might under certain circumstances be executed somewhere in the network stack connecting different (typically homogeneous) computing platforms, in most cases without control of the exact execution by the user. This means that the actual floating-point computation might be performed on the processor, on the network interface card, or even inside a network switch.

5.6.1 The `mpi` port

The `mpi` architecture in `IeeeCC754++` is a proof-of-concept implementation to check network style reductions (cf. Section 1.4.8) for IEEE-conformity. It makes use of the widespread MPI API [MPI15] to trigger floating-point operations inside the network with a master/slave approach in the following manner:

- At the start of `IeeeCC754++`, it is checked that exactly two instances of MPI have been launched via `mpiexec -n 2 IeeeCC754++`.
- Rank 1 (the second `IeeeCC754++` instance) is regarded as the slave process. It runs an infinite loop waiting for messages sent via MPI.

- Rank 0 (the first `IeeeCC754++` instance) is the master process. It executes the usual `IeeeCC754++` initialisation and testing process.
- For every test vector, the master process sends the second operand to the slave process. Afterwards, a reduction operation (i. e. an `MPI_REDUCE`) is triggered on both processes, executing the requested floating-point operation inside the MPI stack.
- When the reduction operation has finished, the master process evaluates the result with the usual `IeeeCC754++` analysis process.
- After the testing loop has been executed, the master process writes the testing summary into the output file, sends a termination message to the slave process, and both processes exit.

Note that the only supported floating-point operators are addition and multiplication in roundTiesToEven mode. Furthermore, the MPI execution (and reduction) model does not allow to control at which point in the MPI stack the actual reduction (i. e. the floating-point operation under consideration) is actually executed. Depending on the interconnect that is used to transport the MPI messages and the actual model of interface card and network switch, the floating-point operation might be executed either in software (with high probability in the MPI layer on the master process), on the network adapter, or on the network switch.

`IeeeCC754++` implements only the `main` FPU inside the `mpi` architecture; it registers the addition and multiplication operators with roundTiesToEven rounding mode. Note that in order to execute a test run, `IeeeCC754++` must be executed via `mpiexec` with exactly two MPI processes, ideally on two nodes connected over a network such as InfiniBand ([IBTA17]) with a network switch supporting direct execution of reduction operations inside the switch (cf. [Mel17]).

5.7 Virtual machines and software libraries

In this section, we discuss architecture ports targeted at user environments that are software-defined, i. e. they do not (directly) depend on the underlying computing hardware. These environments include virtual machine based computation (`Java`, see Section 5.7.1) and software libraries which provide implementations for the floating-point operators done purely in software, such as SoftFloat (Section 5.7.2, MPFR (Section 5.7.3), and CRlibm (Section 5.7.4).

5.7.1 The `java` port

`Java` is "a general-purpose computer programming language [...] specifically designed to have as few implementation dependencies as possible. It is intended to let application developers 'write once, run anywhere' (WORA), meaning that

compiled `Java` code can run on all platforms supporting `Java` without the need for recompilation. `Java` applications are typically compiled to bytecode that can run on any `Java` virtual machine (JVM) regardless of computer architecture. As of 2016, `Java` is one of the most popular programming languages in use, particularly for client-server web applications." [WIK17s] While employing an execution model making use of a virtual machine instead of code compiled to machine instructions leads to outstanding portability, it comes with the penalty of decreased efficiency, especially for numerically intense workloads such as scientific simulations. Therefore, `Java` never gained much momentum in the scientific computing community. Furthermore, `Java` has often been criticised for the design choices regarding its floating-point implementation, see Section 1.4.5 or e. g. [KD98] and [WIK17g].

`IeeeCC754++` nonetheless implements a `java` architecture port since studying the floating-point conformity of such a widespread programming language is important, and it adds support for an interpreted type floating-point implementation to the architectures available in `IeeeCC754++`. However, `Java`'s floating-point support is limited: All operations are executed in roundTiesToEven mode, and exceptions are not implemented. The usual basic arithmetic operations are supported, as well as all conversions with the exception of conversions between floating-point numbers and unsigned integers (due to unsigned integers not being present in `Java`). Note that the conversions have been implemented in `IeeeCC754++` using `Java` style casts since `Java` does not supply specific conversion operators.[6] `Java` supports floating-point numbers in single and double precision.

The `IeeeCC754++ java` architecture supports the mathematical operators available up to `Java` 8. `Java` 9 adds support for an additional mathematical operator, namely `fma`. However, supporting this operator in `IeeeCC754++` poses a serious technical hurdle: Since `Java` does not support conditional compilation, it would not be possible to compile code that makes use of the new `fma` operator with older `Java` versions. If testing the `Java` 9 `fma` is desired, it is possible to implement an `java9` architecture port that requires at least `Java` 9 for compilation. Note that the `fma` routine in the `main` and `strict` FPUs (see below) is implemented with "normal" floating-point operators, i. e. a multiplication followed by an addition.

In order to use `IeeeCC754++`'s testing and analysis facilities, `IeeeCC754++` implements access to `Java`'s native mathematical operators via `Java` JNI (Java Native Interface, [Ora16a]). To achieve this, an instance of a wrapper class providing an interface to the respective native `Java` methods contained in `java.Math` and `java.StrictMath` (see below) is created.

[6] Since `Java` does not support switching the rounding mode, casts (which use the default rounding mode) and conversion routines (which here also use the default rounding mode) yield the same result.

The `strict` FPU

The `java` architecture in `IeeeCC754++` provides two almost identical FPUs: The `main` FPU uses `Java`'s floating-point operators such as + or ∗ as well as casts for conversions between data types. In addition, functions such as square root, remainder, or rounding to integral value are taken from `java.Math`. Note that the `Java` specification does not require methods in `java.Math` to be exactly rounded, but allows for a deviation of up to 2 ulps [Ora16b]. Although these relaxed requirements enable faster execution on some platforms, results need not be bit by bit compatible between different platforms. In order to provide fully portable (and correctly rounded) mathematical operators, the class `java.StrictMath` can be used which guarantees bit-identical results on every virtual machine regardless of the underlying platform. The `strict` FPU inside `IeeeCC754++` makes use of this class. It is almost identical to the `main` FPU, except for those methods for which a `java.StrictMath` version is provided: square root, remainder, and rounding to integral value. Note however that the virtual machines on most platforms call identical underlying hardware routines for the functions in both `java.Math` and `java.StrictMath`, and consequently, testing with the `main` and `strict` FPUs (usually) yield identical results.

5.7.2 The `softfloat` port

Berkeley SoftFloat is "a free, high-quality software implementation of binary floating-point that conforms to the IEEE Standard for Floating-Point Arithmetic" [Hau17] by John Hauser. It supports the most common precisions half, single, double, double extended, and quadruple (cf. Table 1.1) as well as all five rounding modes and all exceptions specified in IEEE 754-2008. Furthermore, all basic arithmetic operations, as well as all conversions with the exception of conversion to and from binary to decimal formats, are implemented.

The `softfloat` architecture in `IeeeCC754++` implements tests for all operations and rounding modes supported by SoftFloat in half, single, and double precision in the `main` FPU. In order to correctly initialise SoftFloat's internal floating-point representations, `IeeeCC754++` implements conversion routines to convert the `C` types `half` (aliased to `int16_t`), **`float`**, and **`double`** to SoftFloat's `float16_t`, `float32_t`, and `float64_t` data types. Note that since support for half precision floating-point numbers is available in SoftFloat starting with version 3b, at least this version of the library must be used.

5.7.3 The `mpfr` port

GNU MPFR [MPFR16] is an efficient arbitrary-precision binary floating-point library with well-defined semantics and correct rounding [Lef11]. It is fully portable in the sense that results are identical on every supported platform. The

four rounding modes and the floating-point exceptions defined in IEEE 754 are supported,[7] as well as all arithmetic functions defined in `C99`. The design principles of MPFR are similar to those of IEEE 754-2008, but extend format support to arbitrary precision. In particular, each operand as well as the result can have their own precision, and the result of an operation will be correctly rounded to target precision. In addition to yielding accurate results, it aims to be fast, i. e. MPFR is as fast as comparable software libraries and even faster in many cases [Fou+08]. MPFR is used in many projects which require correctly rounded results of high accuracy (cf. [MPFR16] for an extensive list). One notable use case is `gcc` which evaluates math functions with constant arguments during compile time by calling respective MPFR functions.

MPFR uses a custom binary format for the representation of floating-point numbers in order to provide arbitrary precision support, making use of GMP [GNU16c]. Subnormal numbers are not supported in the default configuration since MPFR floating-point numbers can be arbitrarily small by design. As a consequence, MPFR is not fully IEEE-conforming in its default configuration. However, it is possible to emulate IEEE 754-2008 support by setting the precision of operands to the values corresponding to e. g. IEEE 754-2008 single and double precision and activating subnormal emulation (which is available in MPFR).

The version of the MPFR library used when building `IeeeCC754++` can be determined by using the `--version` command line argument that is built into the `mpfr` architecture. Finally, note that MPFR supports a fifth rounding mode called "round away from zero" which is different from roundTiesToAway: roundTiesToAway only rounds away from zero when the correct result is halfway between two representable floating-point numbers (rounding to nearest in all other cases), whereas MPFR round away always rounds in the direction opposite of roundTowardZero. Since IEEE 754-2008 does not specify this rounding mode, `IeeeCC754++` does not make use of it.

The `mpfrdef` FPU

In order to verify that MPFR does indeed yield correctly rounded results for the provided operators, `IeeeCC754++` implements two FPUs: In both the `main` and `mpfrdef` FPUs in the `mpfr` architecture, single, double, and quadruple precision numbers are tested by using MPFR floating-point numbers with the precision settings shown in Table 5.3 (cf. also Section 1.2.1). In particular, the precision (length of significand plus hidden bit) has to be specified, as well as minimum and maximum values of the exponent (in order to enable properly working overflow and underflow). `IeeeCC754++` implements all basic arithmetic functions and all conversions, including those from binary to decimal formats and vice versa, as well as support for the four classic rounding modes. There is only one notable difference between the two FPUs: The `main` FPU enables full IEEE 754-2008

[7] Support for the division by zero exception is available in MPFR since version 3.0.0.

support by enabling subnormals support, whereas the `mpfrdef` FPU operates in the default MPFR mode without subnormal numbers.

Format	t	e_{min}	e_{max}
single	24	-148	128
double	53	-1073	1024
quadruple	113	-16493	16384

Table 5.3: *Values that are necessary to enable IEEE 754-2008 support in MPFR. t is the precision in bits, and e_{min} and e_{max} are the minimal and maximal values for the (unbiased) exponent. Note that e_{min} takes the subnormal range into account, i. e. it shows the exponent value for the smallest representable subnormal number. Furthermore, the values of the exponents are shifted by 1 due to MPFR using significands in the range of 0.5 and 1 (compared to significands between 1 and 2 in IEEE 754-2008). For details, cf. [Res16].*

5.7.4 The `crlibm` port

Rounding of the basic arithmetic functions, such as addition, multiplication, or square root, is possible with moderate effort, and bounds for the intermediate precision necessary to correctly round to target precision have been known for a long time. This is reflected (among other things) in the fact that even the first incarnation of IEEE 754 required these functions to be rounded correctly. For the elementary functions, the situation is quite different: For the majority of the possible operands, it is easy to compute correctly rounded results; but due to the Table Maker's Dilemma (see Section 4.1.1) it is very difficult to know (and prove) by how much the precision in intermediate calculations has to be increased in order to provide correctly rounded results for the worst possible cases.

The CRlibm project aims at developing a portable, proven, correctly rounded, and efficient mathematical library for double precision [Dar+06]. Together with thoroughly implemented algorithms, it provides proofs for the correctness of the chosen algorithms and their implementation (see e. g. [Lau08]). CRlibm provides operators only for double precision, but it supports the four classic rounding modes by supplying each operator in four versions (one for each rounding mode). Overflow and underflow exceptions are properly raised, while support for the other exceptions has not been implemented: "Raising the other flags (especially the Inexact flag) is possible but considered too costly for the expected use, and will usually not be implemented." [Dar+06] The following arithmetic functions are supported (for details on these functions, cf. Chapter 4):

- Trigonometric functions and their inverse functions: sine, cosine, tangent, arcsine, arccosine, and arctangent.

- Hyperbolic functions: $\sinh(x)$ and $\cosh(x)$.
- Exponential functions: e^x and $e^x - 1$.
- Logarithmic functions: $\ln(x)$, $\log_2(x)$, $\log_{10}(x)$, and $\ln(1 + x)$.

`IeeeCC754++` provides the `crlibm` architecture which implements these operators inside its `main` FPU. Support for the four rounding modes is achieved by choosing the correct version of the operator under consideration, according to the currently selected rounding mode. Since the mathematical operators in CRlibm support only double operands, `IeeeCC754++` also implements only tests in double precision.

Chapter 6

Selected results

As discussed in the last chapters, `IeeeCC754++` comprises an extensive set of tools to test user environments for IEEE-conformity. Since each user environment consists of a unique combination of computing hardware, operating system, mathematical libraries, and compiler, it is virtually impossible to give a comprehensive overview of the state of IEEE-conformity of current computing environments. Therefore, the main goal of this thesis, as well as this chapter, is not to provide results for as many user floating-point environments as possible, but to provide for a tool set that enables the user to check her own environment. After describing the components of this tool set in the former chapters, we are now able to discuss typical evaluation framework and optimisation framework runs in order to generate conformity results and analyse these. In addition, we present results for a number of selected platforms covering a broad range of different user environments and computer architectures.

In the first section of this chapter, we present a comprehensive example of testing a typical x86 workstation from scratch, including the generation and choice of test vectors, deciding on suitable architectures (in `IeeeCC754++` sense) and FPUs, corresponding evaluation framework runs, and finally analysis of the generated result files with the help of `IeeeCC754++LogViewer`. Afterwards, we (briefly) discuss the role of compilers and show the effect on floating-point results using the former example. The rest of this chapter deals with the results of selected platforms: user environments of some of the most widely spread hardware architectures and FPUs, namely x86 (x87, SSE, AVX) on Xeon and Xeon Phi processors, ARM/AARCH64, and POWER, as well as accelerated devices such as NVidia GPUs. Furthermore, we analyse the IEEE-conformity of software libraries such as SoftFloat and compare the differences of libraries implementing the elementary operators, including `math.h` from `C99`, MPFR, CRlibm, and CUDA. We conclude this chapter with an optimisation framework example and a summary of the results.

It should be noted that although the tools introduced in this thesis can be

of immense help in the process of checking a floating-point environment for IEEE-conformity, it is usually still necessary to inspect the test vectors and the returned results for a conclusive analysis of the mechanisms responsible for errors encountered during the testing process. In other words, although especially the analysis modules of the evaluation framework provide convenient means to summarise different aspects of the testing results, an inspection of the logfiles generated by `IeeeCC754++`, and ultimately an examination of the actual floating-point numbers and exceptions of the errors and their source test vectors cannot be avoided. An example of this process is given in Section 6.1.2.

6.1 A detailed example

Although `IeeeCC754++` and the accompanying tools substantially ease checking the IEEE-conformity of a given user environment, the process of performing the corresponding tests requires a non-negligible number of steps. In this section, we present an extensive example of the complete testing process, including checking the default user environment and FPUs of the underlying platform, i. e. FPUs contained in the actual computer hardware. Furthermore, we take a deeper look at the results and analyse some properties of the floating-point implementations.

6.1.1 User environments

The platform used for this example consists of our (slightly dated) workstation which features a 4th generation Intel Core i7-4770 processor (Haswell microarchitecture, cf. [INT13a]) with openSUSE 13.1 (`x86_64` with Linux kernel `3.12.62-55`). The default compilers provided by openSUSE 13.1 are `gcc` 4.8.1 and `clang` 3.3. Additionally, `clang` version 3.1 to 5.0, `gcc` 4.0 to 7.2, and Intel `icc` 13.0 have been installed onto the workstation.

In this section, we will only use the default compilers. Out of the theoretically possible user environments, we will examine the following:

- The default user environment, i. e. the default architecture (cf. Section 5.1.1) paired with the `gcc` compiler.
- The x86 architecture (i. e. the "native" architecture of the underlying platform) together with the `gcc` and `clang` default compilers. In order to retrieve as much information as possible, the verbose mode (cf. Section 3.3.2) is used.

Other user environments which would be possible on this specific computing platform, which include accelerated environments such as NVidia or OpenCL and software environments such as SoftFloat, MPFR, CRlibm, or `Java`, are not part of this example. For all checks performed in this section, the `t2s` and `t2d` testsets together with the `main` FPUs and the `t3s` and `t3d` testsets with all other FPUs

are used (cf. Section 4.7). In other words, basic operations and conversions are checked in single and double precision. Note that this choice excludes test vectors for the elementary functions from the example. A detailed analysis of results for various architectures supplying the floating-point elementary functions can be found in Section 6.8.

6.1.2 Manual testing procedure

Tests can be executed in a number of different manners: directly using `IeeeCC754++` and the evaluation modules from the evaluation framework to perform every step manually, writing task files for the evaluation framework, or using the supplied `startTests.sh` to specify general settings and execute all necessary tasks. We start by using the first method for the `default` architecture with the `main` FPU, in order to demonstrate how to manually check a user environment if desired, followed by a thorough description of the third approach for the user environments listed above.

Preparations

Since all steps are to be performed manually, no other preparations are needed than the generation of the desired testsets with the script `genUCB.sh`:

```
> cd testsets
> ./genUCB.sh 2 s
Using the following settings:
  type   2
  round
  input  alls
  output t2s

Writing "t2s": add sub mul div rem sqrt fma ct rt b2d d2b i ri rI ru rU ci cI cu
    cU
Done.
> ./genUCB.sh 2 d
Using the following settings:
  type   2
  round
  input  alld
  output t2d

Writing "t2d": add sub mul div rem sqrt fma ct rt b2d d2b i ri rI ru rU ci cI cu
    cU
Done.
```

Listing 6.1: *Generating testsets.*

Building `IeeeCC754++`

The next step is to configure the `IeeeCC754++` source tree for the chosen architecture and FPUs. In this example, we employ the default mode (cf. Section 3.3.3),

i. e. the `default` architecture port together with the verbose output format. In addition to the `main` FPU, we build code for the `generic` and `c99` FPUs.

Note that it is advisable to avoid the configuration and compilation directly inside the source tree. This can e. g. be achieved by using an appropriate subdirectory:

```
> mkdir build
> cd build
> ../configure --enable-arch-default --enable-fpu-generic --enable-fpu-c99
configure: loading site script /usr/share/site/x86_64-unknown-linux-gnu
    ...
configure:

    Build summary:
    --------------

    Build tests?   no
    Use hashing?   yes

    Compilers:     CC=gcc -std=gnu99, CXX=g++ (g++)
    32/64 bit:     native
    Cross compile? no
    Default flags: no

    Modes:         main
    Architecture:  default
    FP units:      generic c99

    Compilation flags:
        CFLAGS:
        CPPFLAGS:
        CXXFLAGS:
        LDFLAGS:
        LIBS:      -lssl -lcrypto


> make
Making all in src
    ...
make[3]: Entering directory '/tmp/example/build/src/common'
  CXX      Bitstring.o
  CXX      DriverFloatRepr.o
  CXX      FP.o
  CXX      FPregistry.o
  CXX      FileOps.o
  CXX      Hex.o
  CXX      IeeeCC754++_util.o
  CXX      Checksum.o
  CXX      Error.o
  AR       libIeeeCC754++.a
  CXX      IeeeCC754_classic.o
  AR       libIeeeCC754classic.a
  CXX      decode.o
  CXXLD    decode
make[3]: Leaving directory '/tmp/example/build/src/common'
Making all in default
make[3]: Entering directory '/tmp/example/build/src/default'
```

```
  CXX      fpenv_default.o
  CXX      fpu_main.o
  CXX      fpu_generic.o
  CXX      fpu_c99.o
  CXX      fpenv_c99.o
../../../src/default/fpenv_c99.cc:110:0: warning: ignoring #pragma STDC
    FENV_ACCESS [-Wunknown-pragmas]
     #pragma STDC FENV_ACCESS ON
 ^
../../../src/default/fpenv_c99.cc:139:0: warning: ignoring #pragma STDC
    FENV_ACCESS [-Wunknown-pragmas]
     #pragma STDC FENV_ACCESS ON
 ^
  CC       fpenv_c99_c.o
  CXX      main_default.o
  CXXLD    IeeeCC754++_default
make[3]: Leaving directory '/tmp/example/build/src/default'
    ...
> ./src/default/IeeeCC754++_default --version
IeeeCC754++ driver executable v0.9.8-dev.
64 bit version compiled for architecture "default" on x86_64 with gcc (4.8.1
    20130909 [gcc-4_8-branch revision 202388]).
Built against revision r983
Built against revision r983
autotools version:
  autoconf: autoconf (GNU Autoconf) 2.69
  automake: automake (GNU automake) 1.13
  m4:       m4 (GNU M4) 1.4.17
```

Listing 6.2: *Configuring and building* `IeeeCC754++`.

Note that after successful compilation, `IeeeCC754++` is called with the parameter `--version` that displays information about the executable.

Executing the tests

To check the IEEE-conformity of the default user environment, we can now execute the necessary tests for the two testsets `t2s` and `t2d`. Note that we employ the `generic` FPU (cf. Section 5.1.1) which includes an `fma` operator not contained in the `main` FPU. The testing results in verbose output format are written into the logfiles `t2s.log` and `t2d.log`.

```
> ./src/default/IeeeCC754++_default -v ../testsets/t2s -f t2s.log --fpu=generic
Calling code for "generic" fpu.
Using logfile: t2s.log
    fma_op:   4410
    fma_err:  257
    inf_op:   1556
    inf_err:  4
    inf_nan_err:  0
    tiny_ops: 3453
    ftz_ops:  0
    ftz_errs: 0
    ulpCount: 152
    ulpLarge: 105
    iulpSum:  180
```

```
    fulpSum:  180
    avgiULPi: 1
    avgiULPf: 1.18422
    avgfULP:  1.18422
    ULPs: 257 (152 with avg: 1.18422; too large: 105)
  [sum] total/err/warn/skip 25059/4082/0/334 v [noftz]
> ./src/default/IeeeCC754++_default -v ../testsets/t2d -f t2d.log --fpu=generic
Calling code for "generic" fpu.
Using logfile: t2d.log
    fma_op:   4527
    fma_err:  302
    inf_op:   1584
    inf_err:  4
    inf_nan_err:  0
    tiny_ops: 3555
    ftz_ops:  0
    ftz_errs: 0
    ulpCount: 197
    ulpLarge: 0
    iulpSum:  225
    fulpSum:  225
    avgiULPi: 1
    avgiULPf: 1.14214
    avgfULP:  1.14214
    ULPs: 197 (197 with avg: 1.14214; too large: 0)
  [sum] total/err/warn/skip 26206/4793/0/334 v [noftz]
```

Listing 6.3: *Executing tests in the default user environment.*

A first coarse analysis can be performed with the console output of the `IeeeCC754++` invocation: The user environment properly supports subnormal numbers, indicated by $\mathcal{O}(3500)$ tiny operations for which no errors occurred. For both precisions, a small number of overflow errors were encountered. Furthermore, a small percentage of `fma` operations resulted in errors, indicating some problem with the `fma` implementation. Interestingly, only 257 test vectors in single precision and 197 test vectors in double precision contained errors in the significand (denoted by the ulp count displayed on the last line), indicating that the other errors are caused by the sign or exceptions being wrong. Note that the 334 skipped test vectors are conversions from and to half precision, cf. Section 4.7.1.

For a deeper analysis of the testing results, the logfiles need to be examined.

Examining the logfiles

The evaluation modules built into the evaluation framework can be utilised without performing a full evaluation framework run (for details, see Section 3.4.1). Listing 6.4 shows the complete output when executing `eval.py` for the single precision logfile `t2s` (the switch `-q` is used to shorten the output generated by `eval.py`).

```
> python eval.py -q ../build/t2s.log
Looking for SQLite and pySQLite.
Try system install (sqlite3)... found SQLite v3.7.17 and pySQLite v2.6.0.
eval.py: IeeeCC754++ job analyser
```

```
>>> 2017-11-06 14:25:07 [debug] Changed debug level to -1.
>>> 2017-11-06 14:25:07 [evalmain] Running on host wmai15 - x86_64(x86_64), Linux
>>> 2017-11-06 14:25:07 [eval] Log file detected, executing standalone mode.
>>> 2017-11-06 14:25:07 [eval] Examining file: t2s.log
>>> 2017-11-06 14:25:07 [eval] t2s.log: Read 24726 lines.
>>> 2017-11-06 14:25:07 [results] Found Summary: total/err/warn/skip
    25059/4082/0/334 v [noftz]
>>> 2017-11-06 14:25:07 [results] SumLine OK.
>>> 2017-11-06 14:25:07 [eval] Eval results:
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
rem       100.00%   1364/1364
sqrt      100.00%   1784/1784
fma        94.33%   4277/4534
b2d         1.63%   8/492
d2b         1.60%   54/3372
i         100.00%   350/350
ri         89.13%   164/184
rI        100.00%   47/47
ru         92.59%   25/27
rU         96.43%   27/28
ci        100.00%   56/56
cI        100.00%   56/56
cu        100.00%   48/48
cU        100.00%   48/48
RESULT     83.49%   20643/24725

   GROUPED

basic     100.00%   12335/12335
extra      96.65%   7425/7682
conv       97.27%   821/844
output      1.60%   62/3864


>>> 2017-11-06 14:25:08 [eval] READ FILES: 1, GOOD FILES: 1   ()
>>> 2017-11-06 14:25:08 [evalmain] Completed run in 0:00:01.
>>> 2017-11-06 14:25:08 [evalmain] Done.
```

Listing 6.4: *Standalone evaluation for single precision.*

As can be seen from the output, the `basic` evaluation function indeed only shows basic analysis information. However, even this output reveals further insight: The execution of all test vectors for the basic operations yields correct results in all rounding modes, with the exception of about 5% of the `fma` test vectors (the `extra` group shows square root, remainder, and `fma` results). Additionally, some of the conversions exhibit rather unexpected behaviour: For the conversions between decimal and binary representation, almost no correct results were returned (operators `b2d` and `d2b`), and there are errors in conversions to integers (`ri`, `ru`, and `rU`).

In order to gain further insight, additional evaluation modules need to be employed. Listing 6.5 shows the output of the `operation_report` evaluation function for all operations, generated with `python eval.py -e operation_report t2s.log`:

```
(Errors, ulps, error count shown)

b2d
    j          0    420
    jd         0     64
        d - different decimal representation
        j - inexact flag not returned
d2b
    emh        0      4
    j          0   3312
    n          0      2
        e - exponent different
        h - result is not an infinity
        j - inexact flag not returned
        m - mantissa different
        n - result is not a NaN
fma
    ma         1     56
    sa         0    105
    xma        1     96
        a - fma error
        m - mantissa different
        s - Different sign
        x - inexact not expected
rU
    x          0      1
        x - inexact not expected
ri
    p          0     20
        p - invalid flag not returned
ru
    p          0      2
        p - invalid flag not returned
```

Listing 6.5: *`operation_report` evaluation function results for `t2s.log`.*

For the `d2b` operator, i. e. for conversion from decimal to binary representation, the vast majority of errors is related to test vectors where the resulting floating-point number is correctly rounded, but the inexact flag was not returned. For the conversion in the other direction, all of the errors missed the inexact flag, with a small number showing also incorrect (decimal) floating-point numbers. All errors concerning rounding to integer formats (`ri`, `ru`, and `rU`) stem from exception errors, meaning that the returned floating-point number is correct.

Concerning the `fma` operation, three types of errors occurred (note that every error in an `fma` test vector is marked as `fma` error and therefore contains the error code "`a`"): operations with incorrect significand, operations with incorrect significand in which the inexact flag was also erroneously returned (which means that the result should have been exact, but was rounded instead), and errors in which only the sign is wrong. The root causes for these phenomena can only be

revealed by further inspection (see below), but there is reason to suspect two different causes: The sign errors might be related to signed zeroes, i. e. the `fma` implementation might not correctly preserve the sign of signed zeroes. The number of errors with incorrect binary representations hints at the possibility that the `fma` operator might not be fused at all, since it roughly corresponds with the number of test vectors expected to yield wrong results when a multiplication is followed by an addition with rounding the intermediate result (about half of the 240 test vectors aimed at this purpose). This suspicion is supported by the fact that the incorrect `fma` test vectors are approximately 1 ulp off (on average).

In order to analyse these small deviations, Listing 6.6 shows the output of the `ulp` evaluation function:

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n           24     14      2      6365
z           38      0      0      6175
u           31      7      1      6090
d           31      7      1      6095
RESULT     124     28      4     24725
```

Listing 6.6: *`ulp` evaluation function results for `t2s.log`.*

The output reveals that the number of errors with incorrect binary representation is slightly larger than the number of errors for `fma`. However, the number of errors with 1 ulp and $2-8$ ulps deviation add up to the exact number of `fma` errors (excluding the sign errors). In order to locate the test vectors causing the errors, it is helpful to generate a testset only containing `fma` test vectors, ideally annotated with the line numbers from the original Coonen test file. Listing 6.7 shows the commands with which such a UCB input file and the corresponding output were generated.

```
> cd build
> ./src/default/IeeeCC754++_default -o fmas -a -s ../src/testsets/fma
> ./src/default/IeeeCC754++_default -v fmas -f fmas.log --fpu=generic
> cd ../eval
> python eval.py -e ulp ../build/fmas.log
```

Listing 6.7: *Generating annotated results for the `fma` testset.*

Listing 6.8 shows the corresponding output file `fmas.log`:

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n           24     14      0      1135
z           38      0      0      1129
u           31      7      0      1135
d           31      7      0      1135
RESULT     124     28      0      4534
```

Listing 6.8: *`ulp` evaluation function results for `fmas.log`.*

First, it can be seen that the deviations up to 8 ulps are indeed related exclusively to the `fma` operation. Additionally, it is now possible to locate the source Coonen vectors that caused the errors by looking either directly at the logfile `fmas.log` or by using output from the `error_list` analysis module. Listing 6.9 contains an excerpt of the errors. It shows the first and last four test vectors with errors related to ulp deviations. Listing 6.10 shows the corresponding lines from the UCB input testset file `fmas`, i. e. lines 51-58, 421-424, and 427-430.

```
[__] l.52 s fma n 3f800001 3fc00000 bf800001 3f000001  => - 3f000002 x  |  xma 1
[__] l.54 s fma z 3f800001 3fc00000 bf800001 3f000001  => - 3f000000 x  |  xma 1
[__] l.56 s fma u 3f800001 3fc00000 bf800001 3f000001  => - 3f000002 x  |  xma 1
[__] l.58 s fma d 3f800001 3fc00000 bf800001 3f000001  => - 3f000000 x  |  xma 1
    ...
[__] l.422 s fma n c3800001 3fb00000 43800001 c2c00002 x => - c2c00000 x  |  ma 2
[__] l.424 s fma d c3800001 3fb00000 43800001 c2c00002 x => - c2c00004 x  |  ma 2
[__] l.428 s fma z c3800001 3fb00000 43800001 c2c00001 x => - c2c00000 x  |  ma 1
[__] l.430 s fma u c3800001 3fb00000 43800001 c2c00001 x => - c2c00000 x  |  ma 1
```

Listing 6.9: *`error_list` evaluation function results excerpt for ulp deviations in `fmas.log`.*

```
! line 210
fmas n eq - 3f800001 3fc00000 bf800001 3f000001
! line 210
fmas z eq - 3f800001 3fc00000 bf800001 3f000001
! line 210
fmas p eq - 3f800001 3fc00000 bf800001 3f000001
! line 210
fmas m eq - 3f800001 3fc00000 bf800001 3f000001
    ...
! line 275
fmas n eq x c3800001 3fb00000 43800001 c2c00002
! line 275
fmas m eq x c3800001 3fb00000 43800001 c2c00002
    ...
! line 276
fmas z eq x c3800001 3fb00000 43800001 c2c00001
! line 276
fmas p eq x c3800001 3fb00000 43800001 c2c00001
```

Listing 6.10: *Excerpt from the UCB input file `fmas`.*

The Coonen test vectors resulting in the observed errors can be found in the original Coonen `fma` file located in `src/testsets/fma` between lines 210 and 276. These lines contain the test vectors which we specifically designed to identify `fma` implementations that are implemented correctly (and distinguish between those that use an intermediate rounding), cf. Section 4.2.1. With this reasoning, we can finally conclude that the errors are caused by the `fma` implementation inside the `default generic` FPU not being fused at all.

Analysing the further contents of the `error_list` analysis module output reveals that the sign errors are indeed related to signed zeros: All errors are caused by multiplications whose correct result is a negative zero due to underflow, i. e. the correct result is too small to be represented by the smallest subnormal number. Furthermore, the errors do not occur with the roundTowardNegative rounding mode. Actually, this behaviour is expected from an implementation with separate multiplication and addition (instead of a fused operator), since the addition of -0 and 0 (in this order) results in a negative zero only in roundTowardNegative mode and a positive zero otherwise. Consequently, the 105 sign errors also reveal that the `fma` implementation is not fused. Listing 6.11 shows the first three corresponding test vectors in the `error_list` analysis function output.

```
[__] l.7510 s fma n 80800000 00800000 00000000 80000000 xu => - 00000000 xu  |  sa 0
[__] l.7512 s fma z 80800000 00800000 00000000 80000000 xu => - 00000000 xu  |  sa 0
[__] l.7514 s fma u 80800000 00800000 00000000 80000000 xu => - 00000000 xu  |  sa 0
```

Listing 6.11: *`error_list` evaluation function results excerpt for sign errors in `fmas.log`.*

Actually, the conclusion that the `fma` operator in the `default generic` FPU is not fused comes at no surprise when taking a look at the actual implementation of the operator (see Listing 6.12 for the single precision code branch): As described in Section 5.1.1, the `fma` operation implemented in the `default main` FPU (which is only accessible via the `default generic` FPU) is simulated by using ordinary multiplication and addition operators.

```
if (isIEEEbinary32())
{
    float res, op1, op2, op3;

    op1 = tofloat();
    op2 = T2.tofloat();
    op3 = T3.tofloat();

    SetLibEnvironment();
    res = op1 * op2 + op3;
    GetLibExceptions();

    DriverFloat_main r(res);
    return r;
}
```

Listing 6.12: *Excerpt from the `fma` operator implementation of the `default generic` FPU.*

Finally, we compare the `fma` testing results gained with the `main` FPU with those of the `c99` FPU. Listing 6.13 shows how these results can be generated with the formerly built `IeeeCC754++` executable and `fmas` UCB input file (cf. Listing 6.7):

```
> cd build
> ./src/default/IeeeCC754++_default -v fmas -f fmas.c99.log
Calling code for "c99" fpu.
Using logfile: fmas.c99.log
    fma_op:   4410
    fma_err:  0
    inf_op:   388
    inf_err:  0
    inf_nan_err:  0
    tiny_ops: 921
    ftz_ops:  0
    ftz_errs: 0
    ulpCount: 0
    ulpLarge: 0
    iulpSum:  0
    fulpSum:  0
    avgiULPi: 0
    avgiULPf: 0
    avgfULP:  0
    ULPs: 0 (0 with avg: 0; too large: 0)
> cd ../eval
> python eval.py ../build/fmas.c99.log
```

Listing 6.13: *Generating results for the `main c99` FPU with the `fma` testset.*

The output of the basic analysis module generated by the `eval.py` invocation can be found in Listing 6.14. It reveals that the `fma` operator contained in the `C99` implementation in the selected default user environment is indeed fused (as expected).

```
(Success rates shown)

fma       100.00%   4534/4534
RESULT    100.00%   4534/4534

   GROUPED

extra     100.00%   4534/4534
```

Listing 6.14: *`basic` evaluation function results for `fmas.c99.log`.*

We conclude the manual testing procedure with a short analysis of operators with double precision operands in the default user environment. Listings 6.15, 6.16, and 6.17 show the output of the `basic`, `operation_report`, and `ulp` analysis modules for the `t2d.log` logfile as generated in Listing 6.3. As can be seen immediately, the types of errors are identical and the number of errors similar to the single precision case. Performing a deeper analysis as shown above reveals that the errors are indeed produced by the same root causes, which means that the single and double precision floating-point operators in this default user environment are implemented in a consistent manner.

```
(Success rates shown)
```

```
add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
rem       100.00%   1364/1364
sqrt      100.00%   1784/1784
fma        93.51%   4349/4651
ct        100.00%   80/80
rt        100.00%   270/270
b2d         1.64%   8/488
d2b         2.06%   84/4072
i         100.00%   350/350
ri         89.13%   164/184
rI        100.00%   47/47
ru         92.59%   25/27
rU         96.43%   27/28
ci        100.00%   48/48
cI        100.00%   56/56
cu        100.00%   40/40
cU        100.00%   48/48
RESULT     81.47%   21079/25872

   GROUPED

basic     100.00%   12335/12335
extra      96.13%   7497/7799
conv       98.05%   1155/1178
output      2.02%   92/4560
```

Listing 6.15: *`basic` evaluation function results for `t2d.log`.*

```
(Errors, ulps, error count shown)

b2d
    j          0    416
    jd         0     64
        d - different decimal representation
        j - inexact flag not returned
d2b
    emh        0      4
    j          0   3980
    n          0      4
        e - exponent different
        h - result is not an infinity
        j - inexact flag not returned
        m - mantissa different
        n - result is not a NaN
fma
    ma         1    101
    sa         0    105
    xma        1     96
        a - fma error
        m - mantissa different
        s - Different sign
        x - inexact not expected
rU
    x          0      1
        x - inexact not expected
ri
    p          0      8
```

```
    xp          0     12
        p - invalid flag not returned
        x - inexact not expected
ru
    p           0      1
    xp          0      1
        p - invalid flag not returned
        x - inexact not expected
```

Listing 6.16: *`operation_report` evaluation function results for `t2d.log`.*

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n            39     14      2      6652
z            38      0      0      6463
u            55      7      1      6374
d            37      7      1      6383
RESULT      169     28      4     25872
```

Listing 6.17: *`ulp` evaluation function results for `t2d.log`.*

6.1.3 (Semi-)Automated testing procedure

In this section, we make use of the evaluation framework to perform IEEE-conformity checks for both environments listed in Section 6.1.1. This example employs the script `startTests.sh` (cf. Section 3.4.3) in order to generate the necessary task files and execute the resulting test runs. Although the graphical application `IeeeCC754++LogViewer` was used in analysing the results, we show excerpts of the logfiles similar to the manual case reported above.

General preparations

As a first step, a number of general choices have to be made, such as which user environments should be tested with which compilers or which testsets should be employed. The environments for this example are described in Section 6.1.1. Before starting the actual testing process, input files according to these settings are needed for `startTests.sh`. These files are created in the directory `hosts/arch/example/` as shown in Listing 6.18:

```
> cd hosts/arch
> mkdir example
> cd example
> touch default.in
> touch x86.in
```

Listing 6.18: *Creating input files for use with `startTests.sh`.*

```
_archname = def
_arch = default
_compilers = gcc
_fpu = main c99
_cores = 4
_testsets = t2s t2d
```

Listing 6.19: *Input file* `default.in`.

```
_archname = x86
_arch = x86
_compilers = gcc clang
_fpu = main
_cores = 4
_testsets = t3s t3d
_cxxflags = -mno-avx
```

Listing 6.20: *Input file* `x86.in`.

Listings 6.19 and 6.20 show the contents of the input files, reflecting the choices for architecture, FPUs, compilers, and testsets. For the meaning of the keywords used, see Table 3.7. The testsets `t2s` and `t2d` were chosen for the `default` architecture to include conversions between binary and decimal conversions and `t3s` and `t3d` for the `x86` architecture to exclude these. To speed up the compilation process, all 4 cores contained in the test workstation are utilised. Note that for the `default` architecture, "`def`" will be used to display results. Also note that in contrast to the manual testing example in Section 6.1.2, we use the `default main` FPU which does not include the (by design faulty) `fma` operator. Finally, the `x86` architecture is compiled with the additional compiler option `-mno-avx`, causing the compiler to emit (legacy) SSE instructions instead of their newer AVX equivalents.

Preparing the evaluation framework

To prepare the evaluation framework for the execution of the desired tests, a local file is needed that tells `startTests.sh` which architectures should be considered when generating evaluation framework input task files and executing the tests. Additionally, a directory has to be supplied containing the task files and can be used by the evaluation framework to place the result logfiles. Listing 6.21 depicts the setup process, while Listing 6.22 shows the `mytests.local` file used by `startTests.sh` to read the corresponding settings.

```
> cd eval
> mkdir ex
> touch mytests.local
```

Listing 6.21: *Creating a setup file for use with* `startTests.sh`.

```
HOST="example"
MYTESTS="def x86"
ARCH="ex"
```

Listing 6.22: *Input file* `mytests.local`.

Note that `HOST` must be set to the directory in `hosts/arch/` containing the input files, `ARCH` denotes the subdirectory in `eval/` containing evaluation framework task and log files, and that the tests specified in `MYTESTS` correspond to the displayed architecture names specified via `_archname` in the input files. Listings 6.23 shows the process of generating the task files from these settings.

```
> ./startTests.sh --refresh
Importing default arch/tests from mytests.local.
Using the following settings:
  HOSTPATH:   /tmp/example/hosts/arch/example
  TARGETPATH: /tmp/example/eval/ex
Continue? (y/n) y
genJobs.py: Starting.
Generating files for arch 'def':
  Generating files for def-gcc:
    Writing file def-gcct2s.test... done.
    Writing file def-gcct2s.eval... done.
    Writing file def-gcct2d.test... done.
    Writing file def-gcct2d.eval... done.
    Writing file def-gcc.job... done.
    Writing file def-gcc.com... done.
    ...done (def-gcc).
    Writing file def.job... done.
...done (def).
genJobs.py: Done.
genJobs.py: Starting.
Generating files for arch 'x86':
  Generating files for x86-gcc:
    Writing file x86-gcct3s.test... done.
    Writing file x86-gcct3s.eval... done.
    Writing file x86-gcct3d.test... done.
    Writing file x86-gcct3d.eval... done.
    Writing file x86-gcc.job... done.
    Writing file x86-gcc.com... done.
    ...done (x86-gcc).
  Generating files for x86-clang:
    Writing file x86-clangt3s.test... done.
    Writing file x86-clangt3s.eval... done.
    Writing file x86-clangt3d.test... done.
    Writing file x86-clangt3d.eval... done.
    Writing file x86-clang.job... done.
    Writing file x86-clang.com... done.
    ...done (x86-clang).
    Writing file x86.job... done.
...done (x86).
genJobs.py: Done.

Generated jobs for the following archs:
   def x86
```

Listing 6.23: *Generating evaluation framework task files.*

Executing the tests

With everything properly set up, the actual tests can be performed with the evaluation framework:

```
> ./startTests.sh
Importing default arch/tests from mytests.local.
Mon Nov  6 18:16:21 CET 2017
Starting test: def
Mon Nov  6 18:16:41 CET 2017
Starting test: x86
Mon Nov  6 18:16:59 CET 2017
```

Listing 6.24: *Running the evaluation framework.*

If testing was successful, the directory `eval/ex/` contains the two logfiles `def.log` and `x86.log`, and `IeeeCC754++LogViewer` can be started in the directory `eval/` to view these logfiles:

```
> ls ex/*.log
ex/def.log  ex/x86.log
> ../tools/IeeeCC754++LogViewer
Using ARCH = ex (from mytests.local).
Using FONT = Adobe Courier (from logviewer.conf).
Using SIZE = 9 (from logviewer.conf).
Reading logfiles...
Starting UI...
Choosing font (size 9):  "Adobe Courier": OK.
```

Listing 6.25: *Results of the evaluation framework run.*

6.1.4 Analysing the logfiles

When generating the task files needed by the evaluation framework in the described way with `startTests.sh`, all test runs will be executed in two testing modes: the normal verbose mode and the verbose mode without regarding exceptions, i. e. the modes `-v` and `-vix` (see Section 3.3.2). This approach enables a quick comparison between the results generated in both modes and therefore easy recognition whether the underlying platform supports exceptions (and to what extent). Furthermore, it reveals how many of the test vectors reported by `IeeeCC754++` as errors resulted in incorrect floating-point numbers being returned (as opposed to missing or superfluous exception flags).

Default environment

As a starting point for the analysis of an evaluation framework run, the end of the corresponding logfile should be examined as it contains summaries of the testing results. Listing 6.26 shows one of these summaries for the default environment (which was tested with `gcc` and the `main` and `c99` FPUs):

```
[eval]  098b8c7e9b583174bf2e9518051a84a1  def-gcct2d_c99_t2d_v           total:  17.43%  25872
     4510  ulp:   857 (    1)  ftz: ???  (    1 out of  3555 zero,     0 errors)
[eval]  649365c18e533ac5cf5c6a7d9e357cac  def-gcct2d_c99_t2d_x           total:   6.60%  25872
     1707  ulp:   857 (    1)  ftz: ???  (    1 out of  3555 zero,     0 errors)
[eval]  c7961890fb119b4ac43c0d209c6aa2b3  def-gcct2s_c99_t2s_v           total:  15.53%  24725
     3840  ulp:   681 (    1)  ftz: no
[eval]  d99b8ab772aeef99f4085de1b743d973  def-gcct2s_c99_t2s_x           total:   6.59%  24725
     1629  ulp:   681 (    1)  ftz: no
[eval]  59afc5309478c9bd2edb70adf0252bb1  def-gcct2d_main_t2d_v          total:  18.60%  25872
     4813  ulp:   197 (    1)  ftz: no
[eval]  7f8bb5456ac4ea07789dab4a5a62310b  def-gcct2d_main_t2d_x          total:   4.46%  25872
     1154  ulp:   197 (    1)  ftz: no
[eval]  824ecfc667e1911e97ddf87944debe7c  def-gcct2s_main_t2s_v          total:  16.57%  24725
     4098  ulp:   152 (    1)  ftz: no
[eval]  0f01a69de40e9832e181ba1de5969a6b  def-gcct2s_main_t2s_x          total:   4.88%  24725
     1207  ulp:   152 (    1)  ftz: no
```

Listing 6.26: *Summary of testing the default architecture.*

The summary contains the following entries: an MD5 hash digest of the corresponding logfile (used for quick recognition of identical logfiles, i. e. identical testing results), a name identifying the test run, the error percentage, the total number of test vectors and the number of test vectors with errors, the total and average deviation in ulps, and additional information e. g. about FTZ and vector errors (as far as such information is available). The trailing `_v` and `_x` in the test run name denote `-v` and `-vix` mode, see above.

In the following, we use a condensed version of this summary as shown in Listing 6.27: the leading "`[eval]`" is omitted, the value of the hash digest is shortened to 8 digits (which should be enough digits to identify identical logfiles in this thesis), and additional information that may be given after the FTZ specification is omitted. The omission is denoted by including the result of the FTZ analysis in square braces. The following values can be displayed: "`ftz: no`" represents the case where nothing was omitted, and "`ftz: [no ]`", "`ftz: [FTZ]`", and "`ftz: [FTS]`" denote the omission of further information such as vector or scalar errors when either FTZ is not used or the command line options `--ftz` or `--ftzsigned` (cf. Section 3.3.6) were given. Finally, the FTZ analysis information, such as number of subnormal results, has been omitted when "`ftz: [yes]`" or "`ftz: [???]`" are shown. When more than the FTZ analysis information is missing, we show the full output in order to analyse the floating-point behaviour of the tested floating-point environment.

```
c3241401  def-gcct2d_c99_t2d_v          total:  17.35%  25872   4490  ulp:   989 (   1)  ftz: [???]
bcad9b8a  def-gcct2d_c99_t2d_x          total:   4.09%  25872   1059  ulp:   989 (   1)  ftz: [???]
57daacd9  def-gcct2s_c99_t2s_v          total:  15.47%  24725   3824  ulp:   827 (   1)  ftz: no
cbc6ab77  def-gcct2s_c99_t2s_x          total:   3.62%  24725    895  ulp:   827 (   1)  ftz: no
6be35c5c  def-gcct2d_main_t2d_v         total:  21.16%  21221   4491  ulp:     0 (   0)  ftz: no
42050d77  def-gcct2d_main_t2d_x         total:   0.34%  21221     72  ulp:     0 (   0)  ftz: no
60590e9c  def-gcct2s_main_t2s_v         total:  18.94%  20191   3825  ulp:     0 (   0)  ftz: no
3f540e87  def-gcct2s_main_t2s_x         total:   0.35%  20191     70  ulp:     0 (   0)  ftz: no
```

Listing 6.27: *Short summary of testing the default architecture.*

In this example, most of the errors are related only to exception handling (denoted by the significantly smaller error rates for the `-vix` runs), and if deviations are found, they are in the order of $\mathcal{O}(1)$ ulp. The error rates for the `c99` FPU are

slightly smaller than for the `main` FPU. Finally, this environment does not employ FTZ, but properly supports subnormal number handling.

For more information, the output of the analysis functions for the different test runs needs to be consulted. For the `-v` runs with the `main` FPU, they are almost identical to the output shown in Section 6.1.2 (since `fma` was not tested here). Listing 6.28 shows that indeed test vectors with binary representation are only returned for conversions between binary and decimal formats. The reasons for this behaviour are discussed in detail in Section 6.1.2.

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
rem       100.00%   1364/1364
sqrt      100.00%   1784/1784
b2d        86.99%   428/492
d2b        99.82%   3366/3372
i         100.00%   350/350
ri        100.00%   184/184
rI        100.00%   47/47
ru        100.00%   27/27
rU        100.00%   28/28
ci        100.00%   56/56
cI        100.00%   56/56
cu        100.00%   48/48
cU        100.00%   48/48
RESULT     99.65%   20121/20191

   GROUPED

basic     100.00%   12335/12335
extra     100.00%   3148/3148
conv      100.00%   844/844
output     98.19%   3794/3864
```

Listing 6.28: *`basic` evaluation function results for the `main` FPU in `-vix` mode with the `t2s` testset.*

Listing 6.29 shows the situation for the `c99` FPU in `-v` mode for single precision test vectors. The main difference to the `main` FPU lies in proper support for `fma` and different errors for the `b2d` operation, cf. Listing 6.30.

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
rem       100.00%   1364/1364
sqrt      100.00%   1784/1784
fma       100.00%   4534/4534
b2d         1.63%   8/492
d2b         1.63%   55/3372
```

```
i          100.00%   350/350
ri          89.13%   164/184
rI         100.00%   47/47
ru          92.59%   25/27
rU          96.43%   27/28
ci         100.00%   56/56
cI         100.00%   56/56
cu         100.00%   48/48
cU         100.00%   48/48
RESULT      84.53%   20901/24725

   GROUPED

basic      100.00%   12335/12335
extra      100.00%   7682/7682
conv        97.27%   821/844
output       1.63%   63/3864
```

Listing 6.29: *`basic` evaluation function results for the `c99` FPU in `-v` mode with the `t2s` testset.*

```
d2b
    em          0      1
    emh         0      1
    j           0   2486
    jm          1    826
    m           1      1
    n           0      2
        e - exponent different
        h - result is not an infinity
        j - inexact flag not returned
        m - mantissa different
        n - result is not a NaN
```

Listing 6.30: *`operation_report` evaluation function excerpt for the `c99` FPU in `-v` mode with the `t2s` testset.*

The `operation_report` reveals that in the decimal to binary conversion routine, a large number of results seem to be incorrectly rounded (denoted by the average deviation from the correct result of 1 ulp). Looking at the `ulp` report output shown in Listing 6.31 supports this assumption. In fact, this type of conversion in the `c99` implementation seems to better support the roundTiesToEven and roundTowardZero rounding modes as compared to roundTowardPositive and roundTowardNegative.

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n             0      0      0      6365
z             0      0      0      6175
u           414      0      1      6090
d           413      0      1      6095
RESULT      827      0      2     24725
```

Listing 6.31: *`ulp` evaluation function excerpt for the `c99` FPU in `-v` mode with the `t2s` testset.*

Further analysis for the `c99` FPU in `-vix` mode, as well as for double precision test vectors, reveals no additional insight. Therefore, we refrain from showing more output.

x86 architecture

For the analysis of the x86 architecture in the selected user environment with the default `gcc` and `clang` compilers, we once again choose one of the summaries at the end of the evaluation framework run logfile `eval/ex/x86.log` as shown in Listing 6.32 as starting point.

```
fa527e35  x86-clangt3d_main_t3d_v         total:   0.22%  16661     37  ulp:     0 (   0)  ftz: no
d2a38004  x86-clangt3d_main_t3d_x         total:   0.01%  16661      1  ulp:     0 (   0)  ftz: no
f1a26455  x86-gcct3d_main_t3d_v           total:   0.14%  16661     23  ulp:     0 (   0)  ftz: no
95846b32  x86-gcct3d_main_t3d_x           total:   0.00%  16661      0  ulp:     0 (   0)  ftz: no
751059c3  x86-clangt3s_main_t3s_v         total:   0.21%  16327     35  ulp:     0 (   0)  ftz: no
edf315f3  x86-clangt3s_main_t3s_x         total:   0.00%  16327      0  ulp:     0 (   0)  ftz: no
2d23461c  x86-gcct3s_main_t3s_v           total:   0.14%  16327     23  ulp:     0 (   0)  ftz: no
71514d10  x86-gcct3s_main_t3s_x           total:   0.00%  16327      0  ulp:     0 (   0)  ftz: no
```

Listing 6.32: *Summary of testing the x86 architecture.*

The most evident difference between the `x86` and `default` architecture runs lies in the significantly lower number of errors. The discrepancy is mainly due to two reasons (notwithstanding a deeper analysis, see below): The testsets `t3s` and `t3d` do not contain test vectors for the conversions between binary and decimal representation, and the `x86 main` FPU does not include an `fma` implementation. These operations (i. e. `b2d`, `d2b`, and `fma`) were responsible for the large majority of errors when checking the default environment.

Further examination of the summary in Listing 6.32 reveals that all errors for `clang` except one are related to exceptions only, implying that the floating-point numbers returned by the `x86 main` in the testing environment are (in principle) all correctly rounded. Another observation from the summary is that error counts for `clang` are slightly higher than for `gcc`. In the following paragraphs, we will once again use output from selected analysis modules to examine the reasons for this behaviour.

We start by looking at the `operation_report` output for `clang` with double precision test vectors which is shown in Listing 6.33 (which incidentally is the only case that shows an error not related to exceptions, see the summary above).

```
cU
    s          0      1
        s - Different sign
rU
    i          0      1
    x          0     13
        i - invalid not expected
        x - inexact not expected
ri
    p          0      8
    xp         0     12
```

```
        p - invalid flag not returned
        x - inexact not expected
ru
    p          0      1
    xp         0      1
        p - invalid flag not returned
        x - inexact not expected
```

Listing 6.33: *`operation_report` output for the `x86` architecture in `-v` mode for `clang` and the `t3d` testset.*

The only floating-point number for which the returned binary value is wrong occurs in the `cU` operator, i. e. converting a 64 bit integer to a double floating-point number, and wrong bits are not in the exponent or significand, but the number has incorrect sign. Listing 6.34 reveals that, strangely enough, the integer 0 is converted into the floating-point number -0 in roundTowardNegative rounding mode.

All other errors are found in conversion from floating-point to integer values and return correct binary representation of the resulting integer, but either raise exception flags that were not expected or miss flags that are necessary.

```
[__] l.27209 d cU d 0000000000000000 0000000000000000 0000000000000000  => - 8000000000000000   |  s
     0
```

Listing 6.34: *`error_list` excerpt for the `x86` architecture in `-v` mode for `clang` and the `t3d` testset.*

Listing 6.35 shows the same analysis function output as Listing 6.33, albeit generated with `gcc`. The types and numbers of errors are mostly identical, except concerning the `cU` error analysed above and 13 test vectors where an inexact or invalid flag was erroneously raised. Comparing the corresponding `error_list` outputs (i. e. comparing the returned floating-point numbers and exceptions; these are not shown here) reveals that the errors produced by `gcc` and `clang` are identical, i. e. both compilers return the same (wrong) exceptions for some of the input test vectors. These are shown in Listing 6.36; all errors occur for the same combination of parameters: test vectors for the `rU` operation (rounding floating-point numbers to 64 bit integers) in roundTowardZero rounding mode for which floating-point numbers exactly representable as integers cause an inexact (and in one case an invalid flag) to be raised. Additionally, `clang` generates a few more errors as discussed before.

```
rU
    x          0      1
        x - inexact not expected
ri
    p          0      8
    xp         0     12
        p - invalid flag not returned
        x - inexact not expected
ru
```

```
    p           0      1
    xp          0      1
        p - invalid flag not returned
        x - inexact not expected
```

Listing 6.35: *`operation_report` output for the `x86` architecture in `-v` mode for `gcc` and the `t3d` testset.*

```
[__] l.26990 d rU z 3ff0000000000000 0000000000000000 0000000000000001  => - 0000000000000001 x  |
     x 0
[__] l.26991 d rU z 4000000000000000 0000000000000000 0000000000000002  => - 0000000000000002 x  |
     x 0
[__] l.26992 d rU z 4008000000000000 0000000000000000 0000000000000003  => - 0000000000000003 x  |
     x 0
[__] l.26993 d rU z 4030000000000000 0000000000000000 0000000000000010  => - 0000000000000010 x  |
     x 0
[__] l.26994 d rU z 4030000000000000 0000000000000000 0000000000000010  => - 0000000000000010 x  |
     x 0
[__] l.26995 d rU z 4070000000000000 0000000000000000 0000000000000100  => - 0000000000000100 x  |
     x 0
[__] l.26996 d rU z 40f0001000000000 0000000000000000 0000000000010001  => - 0000000000010001 x  |
     x 0
[__] l.26997 d rU z 40efffe000000000 0000000000000000 000000000000ffff  => - 000000000000ffff x  |
     x 0
[__] l.27003 d rU z 43effffffffffff 0000000000000000 fffffffffffff800  => - fffffffffffff800 i  |
     i 0
[__] l.27013 d rU z 41d0000000400000 0000000000000000 0000000040000001  => - 0000000040000001 x  |
     x 0
[__] l.27014 d rU z 4340000000000001 0000000000000000 0020000000000002  => - 0020000000000002 x  |
     x 0
[__] l.27015 d rU z 433fffffffffffff 0000000000000000 001fffffffffffff  => - 001fffffffffffff x  |
     x 0
[__] l.27016 d rU z 4330000000000001 0000000000000000 0010000000000001  => - 0010000000000001 x  |
     x 0
```

Listing 6.36: *Another `error_list` excerpt for the `x86` architecture in `-v` mode for `gcc` and the `t3d` testset.*

A similar analysis for the single precision case is left out here since it reveals identical behaviour, i. e. `gcc` and `clang` produce identical errors in conversion to integer operators, and additionally `clang` signals exceptions in a few more cases for rounding to 64 bit integers vectors in roundTowardZero mode.

6.2 Different compilers

As discussed in Section 1.4.6, the compiler plays a vital role in the selection of the execution points of floating-point operations for a given user environment, e. g. choosing hardware routines or software libraries. Furthermore, some features may be implemented directly in the compiler or incorporated into it via libraries, e. g. for input/output or evaluation of constant expressions.

In this section, we take a look at the results of IEEE-conformity testing on the x86 workstation described in Section 6.1.1. Note that the influence of using different compiler options is postponed to Section 6.9, in which the optimisation framework is used to study the interaction of performance and IEEE-conformity with regard to selected compiler switches.

In order to examine the differences between compilers and compiler versions, the tests in this section are performed with the `x86 main` FPU which is identical to the `default main` FPU and almost identical to the `default generic` FPU tested in the last section, except for the missing `fma` operator (which is a "fake" `fma` operator, cf. Section 5.1.1 and Listing 6.12), with the following compilers: GNU `gcc` versions 4.0, 4.7, 4.8, and 7.2, `clang` versions 3.1, 3.8, 4.0, and 5.0, Intel `icc` 13.0.0, and PathScale EkoPath 5.0.5. The corresponding evaluation framework input file is shown in Listing 6.37. Note that we use the `t2` testsets that include binary to decimal conversions in single, double, and extended precision (`t2s`, `t2d`, and `t2l`).

```
_archname = com
_arch = x86
_compilers = gcc-4.0 gcc-4.7 gcc-4.8 gcc-7.2 clang-3.1 clang-3.8 clang-4.0
    clang-5.0 icc-13.0.0 path-5.0.5
_fpu = main
_cores = 8
_testsets = t2s t2d t2l
```

Listing 6.37: *Input file `compilers.in`.*

We once again show the summary of the testing process with the different compilers and versions able to be found at the end of the evaluation framework run logfile. Note that the summary is sorted according to the MD5 hash digest with the following reasoning: When testing results are identical, the corresponding logfiles (and therefore the hash digests) are equal. Sorting by this digest then enables quick recognition of test runs with identical results. Indeed, Listing 6.38 reveals that the four `clang` versions yield identical results for single and double precision, and only the `clang` 3.1 (the oldest version tested here) returns different testing results for extended precision. Comparing the relevant logfiles reveals that `clang` 3.1 returns the exact same floating-point numbers, but for eight test vectors raises an inexact flag when it is not expected. This also explains why error counts are identical between all `clang` compiler version for the `-vix` output.

```
71e23287  testl-gcc48t1d_x87_t1d_v       total:   2.49%  15483    385  ulp:   190 (   1)  ftz: [???]
8b4c00ed  testl-gcc48t1d_x87_t1d_x       total:   1.28%  15483    198  ulp:   190 (   1)  ftz: [???]
023c3fff  com-path505t2s_main_t2s_x      total:   1.05%  20191    212  ulp:   129 (   1)  ftz: no
0264f597  com-gcc40t2d_main_t2d_v        total:  21.19%  21221   4496  ulp:     0 (   0)  ftz: [???]
3156d53c  com-gcc40t2d_main_t2d_x        total:   0.36%  21221     77  ulp:     0 (   0)  ftz: [???]
3d42297e  com-path505t2d_main_t2d_v      total:  39.41%  21221   8364  ulp:   440 (   1)  ftz: [???]
42e00aa4  com-clang31t2d_main_t2d_x      total:   0.34%  21221     73  ulp:     0 (   0)  ftz: no
42e00aa4  com-clang38t2d_main_t2d_x      total:   0.34%  21221     73  ulp:     0 (   0)  ftz: no
42e00aa4  com-clang40t2d_main_t2d_x      total:   0.34%  21221     73  ulp:     0 (   0)  ftz: no
42e00aa4  com-clang50t2d_main_t2d_x      total:   0.34%  21221     73  ulp:     0 (   0)  ftz: no
4a592119  com-gcc47t2l_main_t2l_v        total:  12.57%  19501   2451  ulp:     0 (   0)  ftz: no
4a592119  com-gcc48t2l_main_t2l_v        total:  12.57%  19501   2451  ulp:     0 (   0)  ftz: no
4a592119  com-gcc72t2l_main_t2l_v        total:  12.57%  19501   2451  ulp:     0 (   0)  ftz: no
4c285afd  com-icc1300t2l_main_t2l_v      total:  12.73%  19501   2482  ulp:     0 (   0)  ftz: no
594fff7b  com-icc1300t2l_main_t2l_x      total:   0.06%  19501     12  ulp:     0 (   0)  ftz: no
6b09a654  com-path505t2d_main_t2d_x      total:   2.54%  21221    538  ulp:   440 (   1)  ftz: [???]
752250dd  com-clang38t2l_main_t2l_x      total:   0.06%  19501     12  ulp:     0 (   0)  ftz: no
752250dd  com-clang40t2l_main_t2l_x      total:   0.06%  19501     12  ulp:     0 (   0)  ftz: no
752250dd  com-clang50t2l_main_t2l_x      total:   0.06%  19501     12  ulp:     0 (   0)  ftz: no
7535249c  com-gcc47t2d_main_t2d_x        total:   0.34%  21221     72  ulp:     0 (   0)  ftz: no
7535249c  com-gcc48t2d_main_t2d_x        total:   0.34%  21221     72  ulp:     0 (   0)  ftz: no
7535249c  com-gcc72t2d_main_t2d_x        total:   0.34%  21221     72  ulp:     0 (   0)  ftz: no
```

```
7b5b853b  com-gcc40t2s_main_t2s_v        total:  18.98%  20191   3832  ulp:     0 (   0)  ftz: no
7cbef488  com-gcc40t2s_main_t2s_x        total:   0.38%  20191     77  ulp:     0 (   0)  ftz: no
86dbe53a  com-icc1300t2d_main_t2d_x      total:   0.34%  21221     72  ulp:     0 (   0)  ftz: no
8f3ff57f  com-path505t2l_main_t2l_v      total:  45.25%  19501   8825  ulp:     0 (   0)  ftz: no
9c85512b  com-clang38t2l_main_t2l_v      total:  12.57%  19501   2452  ulp:     0 (   0)  ftz: no
9c85512b  com-clang40t2l_main_t2l_v      total:  12.57%  19501   2452  ulp:     0 (   0)  ftz: no
9c85512b  com-clang50t2l_main_t2l_v      total:  12.57%  19501   2452  ulp:     0 (   0)  ftz: no
a54cf742  com-gcc40t2l_main_t2l_x        total:   0.09%  19501     17  ulp:     0 (   0)  ftz: [???]
b16b11a8  com-gcc40t2l_main_t2l_v        total:  12.59%  19501   2456  ulp:     0 (   0)  ftz: [???]
be26c1b4  com-gcc47t2s_main_t2s_x        total:   0.35%  20191     70  ulp:     0 (   0)  ftz: no
be26c1b4  com-gcc48t2s_main_t2s_x        total:   0.35%  20191     70  ulp:     0 (   0)  ftz: no
be26c1b4  com-gcc72t2s_main_t2s_x        total:   0.35%  20191     70  ulp:     0 (   0)  ftz: no
bfaff2ae  com-clang31t2d_main_t2d_v      total:  21.23%  21221   4505  ulp:     0 (   0)  ftz: no
bfaff2ae  com-clang38t2d_main_t2d_v      total:  21.23%  21221   4505  ulp:     0 (   0)  ftz: no
bfaff2ae  com-clang40t2d_main_t2d_v      total:  21.23%  21221   4505  ulp:     0 (   0)  ftz: no
bfaff2ae  com-clang50t2d_main_t2d_v      total:  21.23%  21221   4505  ulp:     0 (   0)  ftz: no
cf548995  com-clang31t2l_main_t2l_x      total:   0.06%  19501     12  ulp:     0 (   0)  ftz: no
cfa0a6b6  com-gcc47t2l_main_t2l_x        total:   0.06%  19501     12  ulp:     0 (   0)  ftz: no
cfa0a6b6  com-gcc48t2l_main_t2l_x        total:   0.06%  19501     12  ulp:     0 (   0)  ftz: no
cfa0a6b6  com-gcc72t2l_main_t2l_x        total:   0.06%  19501     12  ulp:     0 (   0)  ftz: no
d2bf5b8a  com-path505t2l_main_t2l_x      total:   0.04%  19501      8  ulp:     0 (   0)  ftz: no
d9363a0c  com-icc1300t2d_main_t2d_v      total:  21.29%  21221   4519  ulp:     0 (   0)  ftz: no
da9e546c  com-icc1300t2s_main_t2s_x      total:   0.35%  20191     70  ulp:     0 (   0)  ftz: no
e10893d8  com-path505t2s_main_t2s_v      total:  34.98%  20191   7062  ulp:   129 (   1)  ftz: no
e87ac0fe  com-gcc47t2d_main_t2d_v        total:  21.16%  21221   4491  ulp:     0 (   0)  ftz: no
e87ac0fe  com-gcc48t2d_main_t2d_v        total:  21.16%  21221   4491  ulp:     0 (   0)  ftz: no
e87ac0fe  com-gcc72t2d_main_t2d_v        total:  21.16%  21221   4491  ulp:     0 (   0)  ftz: no
ebf54c0c  com-clang31t2l_main_t2l_v      total:  12.51%  19501   2440  ulp:     0 (   0)  ftz: no
ed76b8a3  com-clang31t2s_main_t2s_v      total:  19.00%  20191   3837  ulp:     0 (   0)  ftz: no
ed76b8a3  com-clang38t2s_main_t2s_v      total:  19.00%  20191   3837  ulp:     0 (   0)  ftz: no
ed76b8a3  com-clang40t2s_main_t2s_v      total:  19.00%  20191   3837  ulp:     0 (   0)  ftz: no
ed76b8a3  com-clang50t2s_main_t2s_v      total:  19.00%  20191   3837  ulp:     0 (   0)  ftz: no
f04a55f4  com-gcc47t2s_main_t2s_v        total:  18.94%  20191   3825  ulp:     0 (   0)  ftz: no
f04a55f4  com-gcc48t2s_main_t2s_v        total:  18.94%  20191   3825  ulp:     0 (   0)  ftz: no
f04a55f4  com-gcc72t2s_main_t2s_v        total:  18.94%  20191   3825  ulp:     0 (   0)  ftz: no
f33cb833  com-icc1300t2s_main_t2s_v      total:  19.08%  20191   3852  ulp:     0 (   0)  ftz: no
fe591597  com-clang31t2s_main_t2s_x      total:   0.35%  20191     70  ulp:     0 (   0)  ftz: no
fe591597  com-clang38t2s_main_t2s_x      total:   0.35%  20191     70  ulp:     0 (   0)  ftz: no
fe591597  com-clang40t2s_main_t2s_x      total:   0.35%  20191     70  ulp:     0 (   0)  ftz: no
fe591597  com-clang50t2s_main_t2s_x      total:   0.35%  20191     70  ulp:     0 (   0)  ftz: no
```

Listing 6.38: *Summary of testing different compilers.*

From the summary, the following can also be concluded:

- The newer `gcc` versions 4.7, 4.8 and 7.2 behave identically for all three testsets. This is noteworthy since the default execution unit chosen by the compiler changed between version 4.7 and 4.8: Where `gcc` 4.7 targeted the SSE unit, `gcc` 4.8 by default emits AVX instructions. This means that in the default environment of the tested platform, `gcc` 4.7 and newer provide consistent results regardless of the underlying FPU (SSE or AVX) being employed.

- `gcc` 4.0 shows slightly higher error counts for all three precisions. Comparing the output files in verbose format reveals that a few errors in the `d2b` operator are fixed in the newer releases.

- Most of the errors returned by all compilers are related to exceptions only, i. e. overall, the returned floating-point numbers are correctly rounded.

- Error counts are comparable for `clang`, `gcc`, and `icc`, whereas the (older) EkoPath compiler generates higher error counts (sometimes higher by a factor of 2). For single and double precision, `gcc` yields the fewest errors, followed by `clang` and `icc` (in this order). For extended precision, `clang` and

`gcc` produce similar error counts, directly followed by `icc`. Again, EkoPath produces a significantly higher error rate.

Once again using the output of the different analysis modules, especially the `basic`, `operation_report`, `roundings`, and `ulp` evaluation functions, reveals the following deficiencies in the results returned by the different compilers:

- For all compilers, most of the errors are generated for input and output of floating-point numbers, i. e. for conversions between binary and decimal formats. Almost all of these are related to exceptions only (mostly the inexact flag not being returned). Additionally, all compilers return (a few) errors only related to exceptions for conversions from integer into floating-point formats.

  Note that the quality of returned test vectors for the operators `d2b` and `b2d` is comparable between `clang`, `gcc`, and `icc`, whereas EkoPath generates significantly more errors (wrong return values) for these operations.

- Wrong result values are almost exclusively returned by the conversions between binary and decimal format, the exception being `clang` and EkoPath which return one test vector with incorrect sign for `cU` in double precision (see the analysis in Section 6.1.2).

- `clang`, `gcc`, and `icc` return a few test vectors signalling the inexact exception for remainder and square root in extended precision in addition to the invalid exception (which should be and is signalled). Interestingly, these test vectors for `sqrt` return correct exceptions with `clang` 3.1

- In single and double precision, EkoPath returns zeroes with incorrect sign for a small number of test vectors for the remainder operation. Some additional errors were found for this operation in double precision where, instead of the correct result, the first operand was returned.

6.3 x86

In this section, we take a look at the IEEE-conformity of a few x86 platforms. Since the `x86 main` FPU has already been analysed in the former sections, we concentrate on the (hardware) FPUs built into x86 CPUs. We start by discussing test results for the x87, SSE and AVX FPUs on two common platforms before examining AVX-512 results generated on a Xeon Phi (KNL) processor.

6.3.1 Xeon

For the x86 results, we chose the following two environments to be tested with the x86 architecture port (see Section 5.2.1) since they represent typical CPUs which are widely deployed in desktop workstations and servers:

- The Intel Core i7-4770 workstation described in Section 6.1.1.
- An x86 server with an Intel Xeon E5-2620 v4 CPU (Broadwell microarchitecture, cf. [INT16b]) that is running CentOS Linux 7.3.1611 (`x86_64` with Linux kernel `3.10.0-514.26.1.el7.x86_64`).

The platforms are tested with the `t3s` and `t3d` testsets. Additionally, the `t3l` testset is used to test IEEE-conformity of the `x87` FPU. The same set of compilers was used on both platforms, namely `gcc` versions 4.7, 4.8, 4.9, 5.5, 6.4, 7.2, and 8-20171105 (the last being a pre-release version of the upcoming `gcc` 8), and `clang` versions 4.0 and 5.0. Additionally, Intel `icc` 13.0.0 20120731 was tested on the Intel Core workstation.

SSE, AVX

The analysis for the SSE and AVX FPUs can be kept short: All of the executed test vectors returned correct results, i. e. for all variations of the corresponding logical FPUs in the x86 port (`sse`, `ssei`, `sses`, `sse3`, `sse3i`, `sse3s`, `avx`, `avxi`, `avxsse`, and `avxssei`), no errors were encountered. Two conclusions can be drawn from this result: First, the SSE and AVX (hardware) FPUs have been implemented on these processors in an IEEE-conforming way. Second, all tested compilers generate valid SSE and AVX instructions and make full use of the IEEE-conformity of the underlying hardware.

Listing 6.39 shows example summary for the `avx` FPU (which uses inline assembler instructions to call the FPU's floating-point operators) on the Intel Core workstation.

```
af5dd5ff  avx-clang40t3d_avx_t3d_x       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-clang50t3d_avx_t3d_x       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc47t3d_avx_t3d_x         total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc48t3d_avx_t3d_x         total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc49t3d_avx_t3d_x         total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc55t3d_avx_t3d_x         total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc64t3d_avx_t3d_x         total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc72t3d_avx_t3d_x         total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc8t3d_avx_t3d_x          total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-icc130t3d_avx_t3d_x        total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-clang40t3s_avx_t3s_x       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-clang50t3s_avx_t3s_x       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc47t3s_avx_t3s_x         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc48t3s_avx_t3s_x         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc49t3s_avx_t3s_x         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc55t3s_avx_t3s_x         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc64t3s_avx_t3s_x         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc72t3s_avx_t3s_x         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc8t3s_avx_t3s_x          total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-icc130t3s_avx_t3s_x        total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-clang40t3s_avx_t3s_v       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-clang50t3s_avx_t3s_v       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc47t3s_avx_t3s_v         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc48t3s_avx_t3s_v         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc49t3s_avx_t3s_v         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc55t3s_avx_t3s_v         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc64t3s_avx_t3s_v         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc72t3s_avx_t3s_v         total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc8t3s_avx_t3s_v          total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-icc130t3s_avx_t3s_v        total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-clang40t3d_avx_t3d_v       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-clang50t3d_avx_t3d_v       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc47t3d_avx_t3d_v         total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
```

```
ea227caf  avx-gcc48t3d_avx_t3d_v          total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc49t3d_avx_t3d_v          total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc55t3d_avx_t3d_v          total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc64t3d_avx_t3d_v          total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc72t3d_avx_t3d_v          total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc8t3d_avx_t3d_v           total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-icc130t3d_avx_t3d_v         total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
```

Listing 6.39: *Summary of testing the `x86 avx` FPU.*

x87

For the `x87` FPU, the situation is slightly different. Listing 6.40 shows the corresponding summary generated on the Intel Core workstation, albeit with only one version of the different compilers (the results generated by the different versions of `clang`, `gcc`, and `icc` are identical).

```
02c0ecac  x87-clang50t3l_x87_t3l_v        total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
02c0ecac  x87-gcc72t3l_x87_t3l_v          total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
02c0ecac  x87-icc130t3l_x87_t3l_v         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-clang50t3l_x87_t3l_x        total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-gcc72t3l_x87_t3l_x          total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-icc130t3l_x87_t3l_x         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
3ef04163  x87-clang50t3s_x87_t3s_v        total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
3ef04163  x87-gcc72t3s_x87_t3s_v          total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
3ef04163  x87-icc130t3s_x87_t3s_v         total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
74fe87bc  x87-clang50t3s_x87_t3s_x        total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
74fe87bc  x87-gcc72t3s_x87_t3s_x          total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
74fe87bc  x87-icc130t3s_x87_t3s_x         total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
96d92363  x87-clang50t3d_x87_t3d_x        total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
96d92363  x87-gcc72t3d_x87_t3d_x          total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
96d92363  x87-icc130t3d_x87_t3d_x         total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
c5634a66  x87-clang50t3d_x87_t3d_v        total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
c5634a66  x87-gcc72t3d_x87_t3d_v          total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
c5634a66  x87-icc130t3d_x87_t3d_v         total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
```

Listing 6.40: *Summary of testing the `x87` FPU on an Intel Core workstation.*

Again, all three compilers generate identical results. In extended precision (i. e. for the `t3l` testset), no errors were encountered, whereas in single and double precision, a small number of errors were reported. The output of the `basic` and `roundings` evaluation functions shown for the `t3d` testset in Listings 6.41 and 6.42 reveal that the errors occur for multiplication and division in roundTiesToEven mode.

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul        99.70%   5392/5408
div        99.91%   2151/2153
rem       100.00%   1364/1364
sqrt      100.00%   1784/1784
RESULT     99.88%   15465/15483
```

Listing 6.41: *`basic` evaluation function results for the `x87` FPU, `t3d` testset, double intermediate precision.*

```
(Success rates shown)

n          99.56%   4057/4075
z         100.00%   3793/3793
u         100.00%   3806/3806
d         100.00%   3809/3809
RESULT     99.88%   15465/15483
```

Listing 6.42: *`roundings` evaluation function results for the `x87` FPU, `t3d` testset, double intermediate precision.*

Listing 6.43 points at the cause for the 18 reported errors: Due to the extended size of the exponent even in the double precision mode of the x87 FPU (cf. Section 5.2.1, page 174), these test vectors are rounded to zero instead of being rounded to the smallest subnormal number (in magnitude), or are rounded to the smallest normalised number instead of being rounded to the largest subnormal number. This phenomenon can only happen with roundTiesToEven (or roundTiesToAway), since for the directed rounding modes it is clear whether a subnormal or a regular floating-point number (i. e. either zero or the smallest normalised number) must be returned.

```
(Operations, ulps, error count shown)

emft    - exponent different
        - flush to zero detected
        - mantissa different
        - result is normalized number, expected tiny (underflown)
    div         0      2
    mul         0      8
mf      - flush to zero detected
        - mantissa different
    mul         0      8
```

Listing 6.43: *`error_report` evaluation function results for the `x87` FPU, `t3d` testset, double intermediate precision.*

A similar analysis for the `t3s` testset reveals that the 18 errors reported in single precision are caused by the same reasons as in the double precision case.

When explicitly enabling extended precision for intermediate computations in the x87 FPU with the `IeeeCC754++` option `--extended`,[1] the results are different as shown in Listing 6.44.

[1]The default behaviour of `IeeeCC754++`'s `x87` FPU concerning intermediate precision is different from the behaviour of code that is generated by compilers in default configuration. `IeeeCC754++` switches the x87 FPU to intermediate precision corresponding to the test vector currently under inspection, i. e. when executing a double precision test vector, double intermediate precision is used. In contrast, when the intermediate precision is not explicitly set in the user program, all compilers use the native extended format of the x87 FPU for all intermediate results.

```
02c0ecac  x87-clang50t3l_x87_t3l_v        total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
02c0ecac  x87-gcc72t3l_x87_t3l_v          total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
02c0ecac  x87-icc130t3l_x87_t3l_v         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-clang50t3l_x87_t3l_x        total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-gcc72t3l_x87_t3l_x          total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-icc130t3l_x87_t3l_x         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
71e23287  x87-clang50t3d_x87_t3d_v        total:   2.49%  15483    385  ulp:   190 (   1)  ftz: [???]
71e23287  x87-gcc72t3d_x87_t3d_v          total:   2.49%  15483    385  ulp:   190 (   1)  ftz: [???]
71e23287  x87-icc130t3d_x87_t3d_v         total:   2.49%  15483    385  ulp:   190 (   1)  ftz: [???]
8b4c00ed  x87-clang50t3d_x87_t3d_x        total:   1.28%  15483    198  ulp:   190 (   1)  ftz: [???]
8b4c00ed  x87-gcc72t3d_x87_t3d_x          total:   1.28%  15483    198  ulp:   190 (   1)  ftz: [???]
8b4c00ed  x87-icc130t3d_x87_t3d_x         total:   1.28%  15483    198  ulp:   190 (   1)  ftz: [???]
d14c1fc4  x87-clang50t3s_x87_t3s_v        total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
d14c1fc4  x87-gcc72t3s_x87_t3s_v          total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
d14c1fc4  x87-icc130t3s_x87_t3s_v         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
e1a5832a  x87-clang50t3s_x87_t3s_x        total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
e1a5832a  x87-gcc72t3s_x87_t3s_x          total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
e1a5832a  x87-icc130t3s_x87_t3s_x         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
```

Listing 6.44: *Summary of testing the `x87` FPU on an Intel Core workstation, extended intermediate precision.*

Here, no errors were encountered in single and extended precision (`t3s` and `t3l` testsets), whereas in double precision, a small number of errors were reported. Listing 6.45 shows that errors again happen only in the roundTiesToEven rounding mode, and Listing 6.46 reveals that the errors occur in all basic operations except remainder and are typical double-rounding cases where in target precision, the floating-point number closest to the returned value would have been chosen differently.

```
(Success rates shown)

n          95.14%   3877/4075
z         100.00%   3793/3793
u         100.00%   3806/3806
d         100.00%   3809/3809
RESULT     98.72%   15285/15483
```

Listing 6.45: *`roundings` evaluation function results for the `x87` FPU, `t3d` testset, extended intermediate precision.*

```
(Operations, ulps, error count shown)

em      - exponent different
        - mantissa different
    add         1      4
    sqrt        1     28
    sub         1      2
emft    - exponent different
        - flush to zero detected
        - mantissa different
        - result is normalized number, expected tiny (underflown)
    mul         0      8
m       - mantissa different
    add         1      4
    div         1     22
    mul         1     16
    sqrt        1    112
```

```
    sub         1       2
```

Listing 6.46: *`error_report` evaluation function results for the `x87` FPU, `t3d` testset, extended intermediate precision.*

Finally, Listings 6.47 and 6.48 show summaries of the testing results from the second user environment generated on the Xeon server described above. Comparing the (shortened) hash digests shows that the results are identical to those generated on the Intel Core workstation, with only the `icc` results missing.

```
02c0ecac  x87-clang50t3l_x87_t3l_v       total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
02c0ecac  x87-gcc72t3l_x87_t3l_v         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-clang50t3l_x87_t3l_x       total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-gcc72t3l_x87_t3l_x         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
3ef04163  x87-clang50t3s_x87_t3s_v       total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
3ef04163  x87-gcc72t3s_x87_t3s_v         total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
74fe87bc  x87-clang50t3s_x87_t3s_x       total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
74fe87bc  x87-gcc72t3s_x87_t3s_x         total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
96d92363  x87-clang50t3d_x87_t3d_x       total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
96d92363  x87-gcc72t3d_x87_t3d_x         total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
c5634a66  x87-clang50t3d_x87_t3d_v       total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
c5634a66  x87-gcc72t3d_x87_t3d_v         total:   0.12%  15483     18  ulp:     0 (   0)  ftz: [???]
```

Listing 6.47: *Summary of testing the `x87` FPU on an Intel Xeon server, native intermediate precision.*

```
02c0ecac  x87-clang50t3l_x87_t3l_v       total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
02c0ecac  x87-gcc72t3l_x87_t3l_v         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-clang50t3l_x87_t3l_x       total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
04c30c1a  x87-gcc72t3l_x87_t3l_x         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
71e23287  x87-clang50t3d_x87_t3d_v       total:   2.49%  15483    385  ulp:   190 (   1)  ftz: [???]
71e23287  x87-gcc72t3d_x87_t3d_v         total:   2.49%  15483    385  ulp:   190 (   1)  ftz: [???]
8b4c00ed  x87-clang50t3d_x87_t3d_x       total:   1.28%  15483    198  ulp:   190 (   1)  ftz: [???]
8b4c00ed  x87-gcc72t3d_x87_t3d_x         total:   1.28%  15483    198  ulp:   190 (   1)  ftz: [???]
d14c1fc4  x87-clang50t3s_x87_t3s_v       total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
d14c1fc4  x87-gcc72t3s_x87_t3s_v         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
e1a5832a  x87-clang50t3s_x87_t3s_x       total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
e1a5832a  x87-gcc72t3s_x87_t3s_x         total:   0.00%  15483      0  ulp:     0 (   0)  ftz: no
```

Listing 6.48: *Summary of testing the `x87` FPU on an Intel Xeon server, extended intermediate precision.*

6.3.2 Xeon Phi: KNL

Since the KNL Xeon Phi processors are only released as regular CPUs (in contrast to KNC Xeon Phis which are only available as accelerators, cf. Section 5.2), checking the IEEE-conformity of a user environment with such a processor can be performed in a manner identical to regular x86 CPUs. We performed the tests on one of the nodes of the QPACE3 supercomputer (see e. g. [Lam16; JSC17b; GRW17]) featuring Intel Xeon Phi 7210 processors (cf. [INT16a]). The user environment consisted of the `x86` architecture port with the `avx512` and `avx512i` FPUs and was tested with the `t3s` and `t3d` testsets with `gcc` versions 4.9, 5.5, 6.4, 7.2, and 8-20171105, and clang version 4.0 and 5.0.

Similar to the AVX units tested before, the analysis for the AVX-512 FPUs can be kept short: Listing 6.49 shows that for the `avx512` FPUs in the KNL CPU,

no errors were encountered. Output for the `avx512i` has been left out since it shows identical errors counts.

```
af5dd5ff  avx-clang40t3d_avx512_t3d_x     total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-clang50t3d_avx512_t3d_x     total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc49t3d_avx512_t3d_x       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc55t3d_avx512_t3d_x       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc64t3d_avx512_t3d_x       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc72t3d_avx512_t3d_x       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
af5dd5ff  avx-gcc80t3d_avx512_t3d_x       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-clang40t3s_avx512_t3s_x     total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-clang50t3s_avx512_t3s_x     total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc49t3s_avx512_t3s_x       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc55t3s_avx512_t3s_x       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc64t3s_avx512_t3s_x       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc72t3s_avx512_t3s_x       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
cb5e73da  avx-gcc80t3s_avx512_t3s_x       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-clang40t3s_avx512_t3s_v     total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-clang50t3s_avx512_t3s_v     total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc49t3s_avx512_t3s_v       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc55t3s_avx512_t3s_v       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc64t3s_avx512_t3s_v       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc72t3s_avx512_t3s_v       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
d8bb83b7  avx-gcc80t3s_avx512_t3s_v       total:   0.00%  18893      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-clang40t3d_avx512_t3d_v     total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-clang50t3d_avx512_t3d_v     total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc49t3d_avx512_t3d_v       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc55t3d_avx512_t3d_v       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc64t3d_avx512_t3d_v       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc72t3d_avx512_t3d_v       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
ea227caf  avx-gcc80t3d_avx512_t3d_v       total:   0.00%  19352      0  ulp:     0 (   0)  ftz: no
```

Listing 6.49: *Summary of testing the `x86 avx512` FPU on a KNL Xeon Phi.*

6.4 ARM

Due to the different licensing models provided by ARM Holdings, a significant variety of slightly different processors based on ARM ISA exist (cf. Section 5.3). Therefore, it is virtually impossible to show results for a comprehensive selection of ARM based user environments. In this section, we concentrate on chosen platforms which cover different application areas: SBCs (single board computers, see [WIK17z]) which are typically used for development purposes or as embedded computer controllers, and processors aimed at the server space such as the X-Gene line of processors [APM17]. The selection of user environments covers the 32 bit ISAs such as ARMv7-A as well as the 64 bit ARMv8-A ISA.

6.4.1 ARM: VFP, NEON

To inspect the floating-point behaviour of environments based on the 32 bit ARM ISA, we use the following user environments which represent different generations of SBCs:

- A Raspberry Pi SBC (see [RPI17c]), featuring a Broadcom BMC2835 CPU which is based on an ARM1176JZF-S processor core with ARMv6 ISA (cf. [RPI17a] and [ARM05]), running Raspbian (Debian) Wheezy 7.11 with Linux kernel `4.9.36+`.

- Cubieboard 2 [CB17], an Open Source SBC featuring an Allwinner A20 Cortex-A7 processor (see [All15a]). The board runs Debian Jessie 8.0 with a custom Linux kernel version `3.4.98-sun7i+`.
- A Raspberry Pi 3 SBC, featuring a Broadcom BMC2837 CPU (cf. [RPI17b]) that runs Raspbian (Debian) Jessie 8.0 with Linux kernel `4.9.35-v7+`. Note that the BCM2837 chip contains ARMv8-A A53 cores, but is operated in 32 bit mode with an ARMv7-A Linux system.

In the following, we refer to these environments as `RPI`, `CB2`, and `RPI3`. All three environments are tested with the `t2s`, `t2d`, `t3s`, and `t3d` testsets (where applicable), and all FPUs from the `arm` architecture port such as the VFP or NEON units (cf. Section 5.3.1) are covered by at least one of the respective user environments.

ARMv6: Raspberry Pi

The `RPI` was chosen in order to show results of user environment based on a processor with ARMv6 ISA. Listing 6.50 shows testing results generated on the `RPI` which are limited to the `arm main` FPU since the processor features a VFP2 FPU (which is not implemented in `IeeeCC754++`) and no NEON support. The tests were performed with `gcc` 4.6.

```
0e0be180  default-gcc46t3d_main_t3d_v     total:  10.09%  16661   1681  ulp:   690 (   1)  ftz: [???]
2e9ddc4f  default-gcc46t3s_main_t3s_v     total:  10.35%  16327   1690  ulp:   675 (   1)  ftz: no
5e320315  default-gcc46t3d_c99_t3d_v      total:   9.98%  21312   2127  ulp:   142 (   1)  ftz: [???]
adc4b382  default-gcc46t3s_c99_t3s_v      total:   4.84%  20861   1010  ulp:     6 (   1)  ftz: [???]
d2ee6af9  default-gcc46t3d_c99_t3d_x      total:   2.21%  21312    471  ulp:   142 (   1)  ftz: [???]
e9c17b3c  default-gcc46t3s_main_t3s_x     total:   4.13%  16327    675  ulp:   675 (   1)  ftz: no
ef155780  default-gcc46t3d_main_t3d_x     total:   5.56%  16661    927  ulp:   690 (   1)  ftz: [???]
fd58c608  default-gcc46t3s_c99_t3s_x      total:   1.61%  20861    336  ulp:     6 (   1)  ftz: [???]
```

Listing 6.50: *Summary of testing the `arm main` FPU on a `RPI`.*

Most of the errors encountered during testing are related only to exceptions. Listings 6.51 and 6.52 show most of the errors happen in conversions from and to decimal representation and converting to 64 bit integers. Moreover, the `main` FPU seems to call a different implementation of the square root operator compared to the `c99` FPU, whereas on the other hand, the remainder operator shows errors in the `c99` FPU not present in the `main` FPU. The few errors in the `c99 fma` are related to signed zeroes.

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
rem       100.00%   1364/1364
sqrt       62.50%   1115/1784
b2d        64.02%   315/492
```

```
d2b        63.02%   2125/3372
i         100.00%   350/350
ri        100.00%   184/184
rI        100.00%   47/47
ru        100.00%   27/27
rU        100.00%   28/28
ci        100.00%   56/56
cI         94.64%   53/56
cu        100.00%   48/48
cU         93.75%   45/48
RESULT     89.60%   18092/20191
```

Listing 6.51: *`basic` evaluation function results for the `arm main` FPU in `-vix` mode with the `t3s` testset on `RPI`.*

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
rem        76.69%   1046/1364
sqrt      100.00%   1784/1784
fma        99.74%   4522/4534
b2d        64.02%   315/492
d2b        63.02%   2125/3372
i         100.00%   350/350
ri        100.00%   184/184
rI        100.00%   47/47
ru        100.00%   27/27
rU        100.00%   28/28
ci        100.00%   56/56
cI         94.64%   53/56
cu        100.00%   48/48
cU         93.75%   45/48
RESULT     92.88%   22965/24725
```

Listing 6.52: *`basic` evaluation function results for the `arm c99` FPU in `-vix` mode with the `t3s` testset on `RPI`.*

Listing 6.53 shows the `operation_report` evaluation function output for the square root. By analysing this output and the corresponding test vectors, it seems that the square root implementation returns identical values for all four rounding modes while dropping the inexact flag. Listing 6.54 gives a hint that the returned value is the roundTiesToEven result: Error counts are significantly lower for roundTiesToEven compared to the other rounding modes.

ARMv7-A: `CB2` and `RPI3`

In the following, we discuss the VFP and NEON SIMD units which represent an optional extension to the ARMv7-A ISA. Since vector execution for VFP has been deprecated in ARMv7-A (see Section 5.3.1), support for vector operations is optional in processors which implement ARMv7-A. The Allwinner A20 processor

```
sqrt
    jem        1      1
    jm         1    668
        e - exponent different
        j - inexact flag not returned
        m - mantissa different
```

Listing 6.53: *`operation_report` evaluation function excerpt for the `arm main` FPU in `-vix` mode with the `t3s` testset on `RPI`.*

```
(Success rates shown)

n           98.99%   5177/5230
z           85.14%   4296/5046
u           90.09%   4464/4955
d           83.77%   4155/4960
RESULT      89.60%   18092/20191
```

Listing 6.54: *`roundings` evaluation function output for the `arm main` FPU in `-vix` mode with the `t3s` testset on `RPI`.*

used in the `CB2` supports vector execution, so the VFP FPUs could be tested as intended. On the `RPI3` on the other hand, using the `vfp` and `vfpv4` as vector units had to be disabled (by removing the command `setVectorUnit()` from the corresponding `Register()` functions in the FPU implementations, see Appendix B.8.2), since its underlying processor is based on ARMv8-A ISA which does not allow using VFP as a SIMD unit.

VFP v3 We start the inspection of FPUs implemented in processors based on ARMv7-A ISA with the `vfp` and `vfps` FPUs of the `arm` architecture port (both of which implement VFP v3). The FPUs were tested on `CB2` and `RPI3` with `gcc` versions 4.4, 4.6, and 4.7. Listings 6.55 and 6.56 show an overview of testing results (in the latter, results for `gcc` 4.4 and 4.6 have been left out since they are identical to those generated with `gcc` 4.7).

```
06c5bacd  arm-vfp-gcc44t3d_vfp_t3d_x      total:   1.58%  18770    296  ulp:    87 (   1)  ftz: no
06c5bacd  arm-vfp-gcc46t3d_vfp_t3d_x      total:   1.58%  18770    296  ulp:    87 (   1)  ftz: no
06c5bacd  arm-vfp-gcc47t3d_vfp_t3d_x      total:   1.58%  18770    296  ulp:    87 (   1)  ftz: no
5770b1f3  arm-vfp-gcc44t3s_vfps_t3s_v     total:   1.50%  18968    284  ulp:   175 (   1)  ftz: no
5770b1f3  arm-vfp-gcc46t3s_vfps_t3s_v     total:   1.50%  18968    284  ulp:   175 (   1)  ftz: no
5770b1f3  arm-vfp-gcc47t3s_vfps_t3s_v     total:   1.50%  18968    284  ulp:   175 (   1)  ftz: no
7a9e8491  arm-vfp-gcc44t3d_vfp_t3d_v      total:   1.83%  18770    344  ulp:    87 (   1)  ftz: no
7a9e8491  arm-vfp-gcc46t3d_vfp_t3d_v      total:   1.83%  18770    344  ulp:    87 (   1)  ftz: no
7a9e8491  arm-vfp-gcc47t3d_vfp_t3d_v      total:   1.83%  18770    344  ulp:    87 (   1)  ftz: no
7d679e23  arm-vfp-gcc44t3d_vfps_t3d_x     total:   1.71%  19419    333  ulp:   224 (   1)  ftz: no
7d679e23  arm-vfp-gcc46t3d_vfps_t3d_x     total:   1.71%  19419    333  ulp:   224 (   1)  ftz: no
7d679e23  arm-vfp-gcc47t3d_vfps_t3d_x     total:   1.71%  19419    333  ulp:   224 (   1)  ftz: no
91dc793c  arm-vfp-gcc44t3s_vfp_t3s_v      total:   1.84%  18653    344  ulp:    43 (   1)  ftz: no
91dc793c  arm-vfp-gcc46t3s_vfp_t3s_v      total:   1.84%  18653    344  ulp:    43 (   1)  ftz: no
91dc793c  arm-vfp-gcc47t3s_vfp_t3s_v      total:   1.84%  18653    344  ulp:    43 (   1)  ftz: no
af87b5d6  arm-vfp-gcc44t3s_vfp_t3s_x      total:   1.59%  18653    296  ulp:    43 (   1)  ftz: no
af87b5d6  arm-vfp-gcc46t3s_vfp_t3s_x      total:   1.59%  18653    296  ulp:    43 (   1)  ftz: no
```

```
af87b5d6  arm-vfp-gcc47t3s_vfp_t3s_x      total:   1.59%  18653    296  ulp:    43 (   1)  ftz: no
c10ba2ae  arm-vfp-gcc44t3s_vfps_t3s_x     total:   1.50%  18968    284  ulp:   175 (   1)  ftz: no
c10ba2ae  arm-vfp-gcc46t3s_vfps_t3s_x     total:   1.50%  18968    284  ulp:   175 (   1)  ftz: no
c10ba2ae  arm-vfp-gcc47t3s_vfps_t3s_x     total:   1.50%  18968    284  ulp:   175 (   1)  ftz: no
d3d9eff2  arm-vfp-gcc44t3d_vfps_t3d_v     total:   1.71%  19419    333  ulp:   224 (   1)  ftz: no
d3d9eff2  arm-vfp-gcc46t3d_vfps_t3d_v     total:   1.71%  19419    333  ulp:   224 (   1)  ftz: no
d3d9eff2  arm-vfp-gcc47t3d_vfps_t3d_v     total:   1.71%  19419    333  ulp:   224 (   1)  ftz: no
```

Listing 6.55: *Summary of testing the `arm vfp` and `vfps` FPUs on `CB2`.*

```
3b4548ac  arm-vfp-gcc47t3d_vfp_t3d_x      total:   1.61%  18770    302  ulp:   197 (   1)  ftz: no
455e704b  arm-vfp-gcc47t3d_vfp_t3d_v      total:   1.61%  18770    302  ulp:   197 (   1)  ftz: no
5770b1f3  arm-vfp-gcc47t3s_vfps_t3s_v     total:   1.50%  18968    284  ulp:   175 (   1)  ftz: no
5fa60afb  arm-vfp-gcc47t3s_vfp_t3s_v      total:   1.38%  18653    257  ulp:   152 (   1)  ftz: no
7d679e23  arm-vfp-gcc47t3d_vfps_t3d_x     total:   1.71%  19419    333  ulp:   224 (   1)  ftz: no
c10ba2ae  arm-vfp-gcc47t3s_vfps_t3s_x     total:   1.50%  18968    284  ulp:   175 (   1)  ftz: no
d3d9eff2  arm-vfp-gcc47t3d_vfps_t3d_v     total:   1.71%  19419    333  ulp:   224 (   1)  ftz: no
d94c1531  arm-vfp-gcc47t3s_vfp_t3s_x      total:   1.38%  18653    257  ulp:   152 (   1)  ftz: no
```

Listing 6.56: *Summary of testing the `arm vfp` and `vfps` FPUs on `RPI3`.*

The output reveals that the results for the `vfps` FPU are identical in both environments. Only two types of errors occur: The `fma` operator is implemented by calling the `vmla` assembler instruction which is not fused (as shown by the results in Listing 6.57, see also Listing 6.5 and the following discussion). Furthermore, a few test vectors used to check conversion to 32 bit integers are not rounded correctly. The analysis for double precision is comparable (the error count in the conversion to integer case is slightly higher) and therefore not shown here.

```
(Errors, ulps, error count shown)

fma
    ma          1     56
    sa          0    105
    xma         1     96
        a - fma error
        m - mantissa different
        s - Different sign
        x - inexact not expected
ri
    m           1     27
        m - mantissa different
```

Listing 6.57: *`operation_report` evaluation function output for the `arm vfps` FPU mode with the `t3s` testset on `RPI3` and `CB2`.*

For the `vfp` FPU on `RPI3`, the situation is similar to the `vfps` FPU with only `fma` errors being present due to the operator not being fused (conversions are not supported in the `vfp` FPU). However, results for the `CB2` are different as shown in Listing 6.58:

```
(Errors, ulps, error count shown)

div
    l           0      8
        l - underflow not returned
```

```
fma
    ema         0     14
    iea         0     16
    iema        0     53
    ijea        0     16
    ijema       0     20
    ijkmga      0      4
    isema       0     27
    jea         0      1
    jema        0     51
    l           0      8
    ma          0     37
    sma         0      9
    x           0      2
    xma         0      8
    xy          0     14
    xyema       1      4
    xyma        1     32
    xysa        0      1
    xysma       0      3
        a - fma error
        e - exponent different
        g - result is NaN, expected infinity
        i - invalid not expected
        j - inexact flag not returned
        k - overflow flag not returned
        l - underflow not returned
        m - mantissa different
        s - Different sign
        x - inexact not expected
        y - underflow not expected
mul
    l           0     16
        l - underflow not returned
```

Listing 6.58: *`operation_report` evaluation function output for the `arm vfp` FPU with the `t3s` testset on `CB2` in `-vio` mode.*

The output shows that on the `CB2`, the results of executing floating-point operations in the VFP unit change between scalar and SIMD operation. For multiplication and division, a few underflow cases are not reported as such (i. e. the underflow flag is not returned), and the `fma` operator produces a significant number of errors not found in the (non-fused) scalar case.[2] Again, the situation is similar for the double precision testset.

VFP v4 At least for the two user environments tested here, the results for the VFP v4 FPUs are better than the VFP v3 results. Listing 6.59 shows a testing summary excerpt for the results of testing the `vfpv4` and `vfpv4s` FPUs with `gcc` 4.9 and the `t3s` and `t3d` testsets (results generated with `gcc` 4.4, 4.6, and 4.7 on `CB2` and with `gcc` 4.4, 4.6, 4.7, 4.8, 4.9, and `clang` 3.5 and 3.7 on `RPI3` are identical).

[2]Note that the errors for `fma` are not due to the operator not being fused, since it returns correct results for our specially designed test vectors in order to distinguish between fused and non-fused `fma` versions.

```
04ba631a  arm-vfpv4-gcc49t3s_vfpv4_t3s_v   total:   0.00%  18653      0  ulp:     0 (   0)  ftz: no
4217c245  arm-vfpv4-gcc49t3s_vfpv4s_t3s_x  total:   0.14%  18968     27  ulp:    23 (   1)  ftz: no
58deb692  arm-vfpv4-gcc49t3s_vfpv4s_t3s_v  total:   0.14%  18968     27  ulp:    23 (   1)  ftz: no
5fe2e292  arm-vfpv4-gcc49t3s_vfpv4_t3s_x   total:   0.00%  18653      0  ulp:     0 (   0)  ftz: no
686349cf  arm-vfpv4-gcc49t3d_vfpv4_t3d_v   total:   0.00%  18770      0  ulp:     0 (   0)  ftz: no
a2387d64  arm-vfpv4-gcc49t3d_vfpv4s_t3d_x  total:   0.16%  19419     31  ulp:    27 (   1)  ftz: no
dafab8d2  arm-vfpv4-gcc49t3d_vfpv4s_t3d_v  total:   0.16%  19419     31  ulp:    27 (   1)  ftz: no
e6c39330  arm-vfpv4-gcc49t3d_vfpv4_t3d_x   total:   0.00%  18770      0  ulp:     0 (   0)  ftz: no
```

Listing 6.59: *Summary of testing the* `arm vfpv4` *and* `vfpv4s` *FPUs on* `RPI3`*.*

For the `vfpv4` FPU, all test vectors could be executed with correct results, whereas for the `vfpv4s` FPU (which includes conversions not contained in the `vfpv4` FPU), the same errors occur as for the `vfps` FPU, see above. Listing 6.60 shows the corresponding `operation_report` output for double precision.

```
(Errors, ulps, error count shown)

ri
    m          1     31
        m - mantissa different
```

Listing 6.60: `operation_report` *evaluation function output for the* `arm vfpv4s` *FPU with the* `t3d` *testset on* `RPI3` *in* `-vio` *mode.*

NEON As with the VFP FPUs discussed above, the different versions of the NEON FPUs implemented in the `arm` architecture port were tested on both `CB2` and `RPI3`. Since results for the four FPUs `neon`, `neonq`, `neoni`, and `neonqi` generated with the different compilers were identical in all cases and in both user environments, we only discuss the `neoni` FPU on the `CB2` together with `gcc` 4.7. NEON only supports single precision operands, so only tests with the `t3s` testset were executed. Listing 6.61 shows the corresponding testing results.

```
ae1de173  neon-ftn-gcc47t3s_neoni_t3s_x   total:  39.14%  14434   5649  ulp:  2110 (   1)  ftz: [yes]
d6c60016  neon-ftn-gcc47t3s_neoni_t3s_v   total:  68.27%  14434   9854  ulp:  2110 (   1)  ftz: [yes]
```

Listing 6.61: *Summary of testing the* `arm neoni` *FPU on* `CB2`*.*

The most notable difference to the FPUs analysed so far is the significantly larger error rate which is at almost 40% when ignoring exception related errors and more than two thirds otherwise. The second striking difference is `IeeeCC754++` claiming that the `neoni` FPU is using FTZ. Since the FTZ information is shortened in the dense output format used in Listing 6.61, we show the unabridged summary in Listing 6.62.

```
[eval]  ae1de1739762baef400060f73f76baa6  neon-ftn-gcc47t3s_neoni_t3s_x   total:  39.14%  14434
     5649  ulp:  2110 (    1)  ftz: yes ( 2320 out of  2320 zero,    24 errors)  >30% ZEROES!
[eval]  d6c60016a4717fd522335215add4a48d  neon-ftn-gcc47t3s_neoni_t3s_v   total:  68.27%  14434
     9854  ulp:  2110 (    1)  ftz: yes ( 2320 out of  2320 zero,    24 errors)  >30% ZEROES!
```

Listing 6.62: *Full summary of testing the* `arm neoni` *FPU on* `CB2`*.*

This output shows that all operations supposed to return subnormal results are flushed to zero, which accounts for about half of the errors in `-vix` mode. Indeed, this is to be expected since by default, NEON flushes subnormal results to zero due to performance reasons (cf. [ARM14]).

Before analysing the cause for the other errors, we discuss the impacts of the NEON unit using FTZ. Listings 6.63 and 6.64 show the full summary for test runs which were executed with the command line options `--ftz` and `--ftzsigned`, respectively.

```
[eval]  57ebd5eb0c0d4b33738f2be0b7f8eb09  neon-ftz-gcc47t3s_neoni_t3s_x   total:  30.58%  14434
     4414  ulp:  2110 (    1)  ftz: FTZ  >30% ZEROES!
[eval]  7e578c60e113f4ad34e270123743eb7c  neon-ftz-gcc47t3s_neoni_t3s_v   total:  63.42%  14434
     9154  ulp:  2110 (    1)  ftz: FTZ  >30% ZEROES!
```

Listing 6.63: *Full summary of testing the `arm neoni` FPU with `-ftz` on `CB2`.*

```
[eval]  869f0c885dd2b613d8a78cd07bd091b8  neon-fts-gcc47t3s_neoni_t3s_x   total:  23.73%  14434
     3425  ulp:  2110 (    1)  ftz: FTS  >30% ZEROES!
[eval]  fda84e0e7bb82e7eb733bd083ec80e63  neon-fts-gcc47t3s_neoni_t3s_v   total:  60.15%  14434
     8682  ulp:  2110 (    1)  ftz: FTS  >30% ZEROES!
```

Listing 6.64: *Full summary of testing the `arm neoni` FPU with `-ftzsigned` on `CB2`.*

As expected, the error numbers decrease, albeit not to the amount that could be expected if either flushing to signed zeroes or $+0$ would have been implemented correctly (which should be on the order of $5649 - 2320 = 3329$ errors in the `-vix` case). Looking at the test vectors reveals that indeed NEON flushes to signed zero, but does not always set the correct sign.

Due to the large error counts, an in-depth analysis of all errors produced by the NEON units is beyond the scope of this thesis. However, looking at the output of some of the evaluation functions gives further insight. As a first observation, exceptions do not seem to be well supported in the NEON FPUs as can be seen from the error rate decreasing by almost 30% when ignoring errors only related to exception flags. Second, Listing 6.65 shows that support for the roundTiesToEven rounding mode is significantly better than for the other modes. Finally, according to the excerpt from the `error_report` evaluation function output given in Listing 6.66 (which shows results for the error combination that is responsible for the largest number of errors), a significant number of results is computed almost correctly, but rounded to the wrong floating-point neighbour.

6.4.2 AARCH64: ASIMD, SVE

ARMv8-A: XGene, XGene2, Pine64

In order to generate results for `IeeeCC754++`'s `aarch64` architecture (which is targeted at ARMv8-A ISA), we used three user environments:

```
(Success rates shown)

n          85.16%   3241/3806
z          73.39%   2606/3551
u          74.75%   2644/3537
d          71.13%   2518/3540
RESULT     76.27%   11009/14434
```

Listing 6.65: *`roundings` evaluation function output for the `arm neoni` FPU with `-ftzsigned` in `-vix` mode with the `t3s` testset on `CB2`.*

```
jm      - inexact flag not returned
        - mantissa different
    add         1    362
    ci          1      3
    cu          1      3
    div         1    138
    mul         1    350
    ri          1     27
    sqrt        1    817
    sub         1    181
```

Listing 6.66: *`error_report` evaluation function excerpt for the `arm neoni` FPU with `-ftzsigned` in `-vix` mode with the `t3s` testset on `CB2`.*

- The X-Gene processor designed by Applied Micro represents one of the first available server processors based on ARMv8-A ISA. The user environment used for the tests consists of an APM883208-X1 processor which is located on an X-Gene X-C1 evaluation board (cf. [APMa]). It runs Ubuntu Linux 14.04.4 LTS with kernel version `3.13.0-85-generic`; the compiler installed on this board is `gcc` 4.8.4.
- The second user environment consists of an Applied Micro X-Gene X-C2 evaluation board (see [APMb]) containing an APM883408-X2 processor, which is an enhanced newer version of the X-Gene processor. This X-Gene 2 server environment runs Ubuntu 16.04.2 LTS with Linux kernel version `4.4.0-38.57-generic` and features `gcc` 5.4.1.
- Finally, we used an early AARCH64 SBC called Pine64 (cf. [Pin17]) as a user environment. In contrast to the X-Gene boards (which feature custom CPU cores based on ARMv8-A ISA), the Pine64 boards contains an Allwinner A64 processor (see [All15b]) with standard Cortex-A53 cores. The test environment runs Linux kernel `3.10.102-0-pine64-longsleep` on Debian Jessie version 8.9. We tested this system with `gcc` 4.8 and 4.9 and `clang` 3.5.

All three environments were tested with the `t3s` and `t3d` testsets.

ASIMD In contrast to earlier versions of the ARM ISA, the initial release of ARMv8-A ISA featured only one execution unit targeted at floating-point calculations, and this ASIMD FPU is mandatory (for the newly introduced SVE FPU, see below). We shortly remind of the naming of the logical FPUs implementing ASIMD in `IeeeCC754++`: The `asimd` FPU features an inline assembler implementation whereas the `neoni` and `neonqi` FPUs use compiler intrinsics (for more details, see Section 5.3.2).

We start the analysis of the ASIMD FPU with results generated on X-Gene 2 and briefly discuss differences to the tests executed on the X-Gene and Pine64 boards afterwards. Listing 6.67 shows the testing summary for X-Gene 2 with `gcc` 5.4 and the `t3s` and `t3d` testsets.

```
14afe7cb  aarch64-gcct3s_neonqi_t3s_v     total:   1.27%  19318    245  ulp:    23 (   1)  ftz: no
49a8061f  aarch64-gcct3s_main_t3s_x       total:   0.00%  16327      0  ulp:     0 (   0)  ftz: no
6fd9b3a6  aarch64-gcct3s_asimd_t3s_v      total:   1.13%  19318    218  ulp:     0 (   0)  ftz: no
731c2545  aarch64-gcct3d_asimd_t3d_v      total:   1.11%  19649    218  ulp:     0 (   0)  ftz: no
731c2545  aarch64-gcct3d_neonqi_t3d_v     total:   1.11%  19649    218  ulp:     0 (   0)  ftz: no
749da462  aarch64-gcct3d_main_t3d_v       total:   0.12%  16661     20  ulp:     0 (   0)  ftz: no
7e04c619  aarch64-gcct3s_neonqi_t3s_x     total:   0.14%  19318     27  ulp:    23 (   1)  ftz: no
7f745106  aarch64-gcct3d_neoni_t3d_x      total:   0.00%  19299      0  ulp:     0 (   0)  ftz: no
800ff616  aarch64-gcct3d_asimd_t3d_x      total:   0.00%  19649      0  ulp:     0 (   0)  ftz: no
800ff616  aarch64-gcct3d_neonqi_t3d_x     total:   0.00%  19649      0  ulp:     0 (   0)  ftz: no
b1be2119  aarch64-gcct3d_main_t3d_x       total:   0.00%  16661      0  ulp:     0 (   0)  ftz: no
c694cff0  aarch64-gcct3s_neoni_t3s_v      total:   1.27%  19318    245  ulp:    23 (   1)  ftz: no
d13e2dd2  aarch64-gcct3s_neoni_t3s_x      total:   0.14%  19318     27  ulp:    23 (   1)  ftz: no
d6214873  aarch64-gcct3s_main_t3s_v       total:   0.12%  16327     20  ulp:     0 (   0)  ftz: no
eac0e90d  aarch64-gcct3d_neoni_t3d_v      total:   1.13%  19299    218  ulp:     0 (   0)  ftz: no
f420a398  aarch64-gcct3s_asimd_t3s_x      total:   0.00%  19318      0  ulp:     0 (   0)  ftz: no
```

Listing 6.67: *Summary of testing results on X-Gene 2.*

A few observations are striking: In contrast to the NEON units in ARMv7-A, the ASIMD unit supports operations in double precision and does not use FTZ. Furthermore, the `main` and `asimd` FPUs exhibit only errors related to exceptions. Overall, error counts are quite low, and most of these errors are due to exception handling. Listings 6.68 and 6.69 show the errors for the `main` and `asimd` FPUs. In the `main` FPU, some invalid operations are not flagged as such for conversion to integers, and in the `asimd` FPU, the inexact exception was not signalled for a number of test vectors for the round to integral operator.

```
(Errors, ulps, error count shown)

ri
    p           0     20
        p - invalid flag not returned
```

Listing 6.68: *`operation_report` evaluation function output for the `aarch64 main` FPU in `-vio` mode with the `t3s` testset.*

For the `neoni` and `neonqi` FPUs, the errors are almost similar to those of the `asimd` FPU, as can be seen in Listing 6.70. This behaviour is mostly to be expected since the same instructions should be used inside these FPUs (aside from one using inline assembler calls and the other two FPUs using compiler

```
(Errors, ulps, error count shown)

i
    j          0    218
        j - inexact flag not returned
```

Listing 6.69: *`operation_report` evaluation function output for the `aarch64 asimd` FPU in `-vio` mode with the `t3s` testset.*

intrinsics). Therefore, it seems that the `vrdni` intrinsics call used in the `neoni` and `neonqi` FPUs either maps to a different instruction than the `frinti` operator called inside the `asimd` FPU or behaves differently. Interestingly, when looking at the `roundings` evaluation function for the `neoni -vix` results (see Listing 6.71), it seems that an operator which only supports the roundTowardZero rounding mode is used (which is the behaviour of the SVE unit, see below).

```
(Errors, ulps, error count shown)

i
    j          0    218
        j - inexact flag not returned
ri
    m          1     27
        m - mantissa different
```

Listing 6.70: *`operation_report` evaluation function output for the `aarch64 neoni` FPU in `-vio` mode with the `t3s` testset.*

```
(Success rates shown)

n           99.84%   5024/5032
z          100.00%   4767/4767
u           99.79%   4747/4757
d           99.81%   4753/4762
RESULT      99.86%   19291/19318
```

Listing 6.71: *`roundings` evaluation function output for the `aarch64 neoni` FPU in `-vix` mode with the `t3s` testset.*

Once again, we omit results for the double precision testsets, since they are mostly identical.

Listing 6.72 shows the testing results generated on X-Gene with the same settings as used on X-Gene 2.

```
0ec3622e  aarch64-gcct3d_neoni_t3d_x      total:   0.00%    104      0  ulp:     0 (   0)  ftz: no
14afe7cb  aarch64-gcct3s_neonqi_t3s_v     total:   1.27%  19318    245  ulp:    23 (   1)  ftz: no
3c259c85  aarch64-gcct3s_neoni_t3s_x      total:  13.54%  19318   2615  ulp:   370 (   1)  ftz: [???]
49a8061f  aarch64-gcct3s_main_t3s_x       total:   0.00%  16327      0  ulp:     0 (   0)  ftz: no
614bae00  aarch64-gcct3s_neoni_t3s_v      total:  15.02%  19318   2901  ulp:   370 (   1)  ftz: [???]
6fd9b3a6  aarch64-gcct3s_asimd_t3s_v      total:   1.13%  19318    218  ulp:     0 (   0)  ftz: no
```

```
731c2545  aarch64-gcct3d_asimd_t3d_v      total:   1.11%  19649    218  ulp:     0 (   0)  ftz: no
731c2545  aarch64-gcct3d_neonqi_t3d_v     total:   1.11%  19649    218  ulp:     0 (   0)  ftz: no
749da462  aarch64-gcct3d_main_t3d_v       total:   0.12%  16661     20  ulp:     0 (   0)  ftz: no
7e04c619  aarch64-gcct3s_neonqi_t3s_x     total:   0.14%  19318     27  ulp:    23 (   1)  ftz: no
800ff616  aarch64-gcct3d_asimd_t3d_x      total:   0.00%  19649      0  ulp:     0 (   0)  ftz: no
800ff616  aarch64-gcct3d_neonqi_t3d_x     total:   0.00%  19649      0  ulp:     0 (   0)  ftz: no
b1be2119  aarch64-gcct3d_main_t3d_x       total:   0.00%  16661      0  ulp:     0 (   0)  ftz: no
d6214873  aarch64-gcct3s_main_t3s_v       total:   0.12%  16327     20  ulp:     0 (   0)  ftz: no
f420a398  aarch64-gcct3s_asimd_t3s_x      total:   0.00%  19318      0  ulp:     0 (   0)  ftz: no
fe09e1dd  aarch64-gcct3d_neoni_t3d_v      total:   0.00%    104      0  ulp:     0 (   0)  ftz: no
```

Listing 6.72: *Summary of testing results on X-Gene.*

When comparing the checksums for these results with the checksums of the X-Gene 2 results, the only differences are found in the `neoni` FPU. A surprisingly low number of 104 double precision operations were executed, and the error count for single precision vectors is significantly higher. The reason for this behaviour lies in the compiler being used: Full ASIMD support for D type registers (cf. Section 5.3.1) is only available since `gcc` version 4.9, but `gcc` 4.8 was used on X-Gene, with the described two consequences: For double precision, only intrinsics for conversions from 64 bit integers are available, and single precision intrinsics for D type registers do not correctly map to the underlying hardware instructions.

Finally, we show test results for `clang` on AARCH64 generated on the Pine64 SBC. Inspection of the summary in Listing 6.73 reveals that the results are mostly identical to those generated with `gcc` 5.4 on X-Gene 2 with the exception of the `neoni` FPU with the `t3d` testset. Again, only 104 test vectors were executed, and again, this is due to the compiler version: As of `clang` 3.5, intrinsics for double precision operations on D type registers are not available. In contrast to `gcc` 4.8 on X-Gene, single precision works as expected in the `neoni` FPU (i. e. it shows behaviour identical to that on X-Gene 2).

```
14afe7cb  aarch64-gcct3s_neonqi_t3s_v     total:   1.27%  19318    245  ulp:    23 (   1)  ftz: no
49a8061f  aarch64-gcct3s_main_t3s_x       total:   0.00%  16327      0  ulp:     0 (   0)  ftz: no
6fd9b3a6  aarch64-gcct3s_asimd_t3s_v      total:   1.13%  19318    218  ulp:     0 (   0)  ftz: no
731c2545  aarch64-gcct3d_asimd_t3d_v      total:   1.11%  19649    218  ulp:     0 (   0)  ftz: no
731c2545  aarch64-gcct3d_neonqi_t3d_v     total:   1.11%  19649    218  ulp:     0 (   0)  ftz: no
749da462  aarch64-gcct3d_main_t3d_v       total:   0.12%  16661     20  ulp:     0 (   0)  ftz: no
7e04c619  aarch64-gcct3s_neonqi_t3s_x     total:   0.14%  19318     27  ulp:    23 (   1)  ftz: no
7f745106  aarch64-gcct3d_neoni_t3d_x      total:   0.00%  19299      0  ulp:     0 (   0)  ftz: no
800ff616  aarch64-gcct3d_asimd_t3d_x      total:   0.00%  19649      0  ulp:     0 (   0)  ftz: no
800ff616  aarch64-gcct3d_neonqi_t3d_x     total:   0.00%  19649      0  ulp:     0 (   0)  ftz: no
b1be2119  aarch64-gcct3d_main_t3d_x       total:   0.00%  16661      0  ulp:     0 (   0)  ftz: no
c694cff0  aarch64-gcct3s_neoni_t3s_v      total:   1.27%  19318    245  ulp:    23 (   1)  ftz: no
d13e2dd2  aarch64-gcct3s_neoni_t3s_x      total:   0.14%  19318     27  ulp:    23 (   1)  ftz: no
d6214873  aarch64-gcct3s_main_t3s_v       total:   0.12%  16327     20  ulp:     0 (   0)  ftz: no
eac0e90d  aarch64-gcct3d_neoni_t3d_v      total:   1.13%  19299    218  ulp:     0 (   0)  ftz: no
f420a398  aarch64-gcct3s_asimd_t3s_x      total:   0.00%  19318      0  ulp:     0 (   0)  ftz: no
```

Listing 6.73: *Summary of testing results on Pine64.*

Altogether, our testing results for different user environments using ARMv8-A ISA indicate that the floating-point behaviour of ASIMD and overall floating-point accuracy is consistently implemented, even on quite different platforms, and that only a minimal amount of errors have to be expected for conversions to integers. Additionally, it seems that floating-point behaviour in ARMv8-A is much more IEEE-conforming when compared to ARMv7-A.

ARMv8-A: SVE

As of the writing of this thesis, no hardware is yet available which supports the SVE extension to the ARMv8-A ISA. Therefore, we used the ARM instruction emulator `armie` version 1.2-3 (cf. [ARMa]) as the floating-point environment to implement the `aarch64 sve` FPU and generate the following test results. The SVE capable `IeeeCC754++` executable that performed the tests was compiled with ARM `clang` version 1.4 (build 48) which is based on `clang` 5.0.0.

Our implementation supports the VLA programming model by using SIMD vectors of 2048 bytes length (the largest value allowed for SVE hardware realisations) and relying on the VLA capabilities built into the compiler (and the platform) to map the execution of the SIMD operations to the vector length supported by the hardware. In order to verify that our code was properly implemented, we performed all tests with `armie` for all permitted values of vector length, i. e. for the lengths $i \cdot 128$ with $i = 1, \ldots, 16$. All logfiles generated with the different vector sizes were found to be identical.

Listings 6.74 and 6.75 show the output of the `basic` evaluation function of test runs using the `sve` FPU with the `t3s` or `t3d` testsets.

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
sqrt      100.00%   1784/1784
fma       100.00%   4534/4534
i         100.00%   350/350
ri         85.33%   157/184
rI        100.00%   47/47
ru        100.00%   27/27
rU        100.00%   28/28
ci        100.00%   56/56
cI        100.00%   56/56
cu        100.00%   48/48
cU        100.00%   48/48
RESULT     99.86%   19470/19497
```

Listing 6.74: *`basic` evaluation function results for the `arm sve` FPU in `-vio` mode with the `t3s` testset.*

```
add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
sqrt      100.00%   1784/1784
fma       100.00%   4651/4651
ct        100.00%   80/80
rt        100.00%   270/270
i         100.00%   350/350
ri         83.15%   153/184
rI        100.00%   47/47
```

```
ru        100.00%   27/27
rU        100.00%   28/28
ci        100.00%   48/48
cI        100.00%   56/56
cu        100.00%   40/40
cU        100.00%   48/48
RESULT     99.84%   19917/19948
```

Listing 6.75: *`basic` evaluation function results for the `arm sve` FPU in `-vio` mode with the `t3d` testset.*

The output shows that the design and implementation of the instruction emulation of `armie` have been meticulously executed: No errors were encountered for most operations. In fact, errors in the `ri` operator were to be expected due to the `sve` FPU only supporting roundTowardZero mode for conversion to integers (cf. Section 5.3.2). The output of the `error_report` and `roundings` evaluation functions shown in Listings 6.76 and 6.77 conform this suspicion: The errors consist only of incorrectly rounded integers in the roundTiesToEven, roundTowardPositive, and roundTowardNegative rounding modes. Consequently, we can conclude that no errors were encountered for all operators supported by SVE when testing the FPU via the instruction emulator. As soon as actual hardware implementing the SVE extension is released, these tests need to repeated to confirm that the hardware meets the high implementation standards of the software emulation.

```
(Errors, ulps, error count shown)

ri
    m          1     27
        m - mantissa different
```

Listing 6.76: *`operation_report` evaluation function results for the `arm sve` FPU in `-vio` mode with the `t3s`.*

```
(Success rates shown)

n          99.84%   5050/5058
z         100.00%   4868/4868
u          99.79%   4773/4783
d          99.81%   4779/4788
RESULT     99.86%   19470/19497
```

Listing 6.77: *`roundings` evaluation function results for the `arm sve` FPU in `-vio` mode with the `t3s`.*

6.5 PowerPC

After presenting some "future" results with the `aarch64 sve` architecture, we discuss some current and some "historic" results in this section. We examine

the floating-point behaviour of a POWER8 system and take a brief look at two incarnations of the discontinued Cell microarchitecture: the original Cell processor contained in the Sony PS 3 and the enhanced PowerXCell 8i processor as deployed in the QPACE supercomputer.

6.5.1 POWER8

The user environment on which the following tests were executed is an IBM POWER System S824L [IBM17] running Ubuntu 16.04.2 LTS with Linux kernel version `4.4.0-45-generic`. The executables were compiled with `gcc` 5.4.0 and `XLC` 13.1.5; testsets `t3s` and `t3d` were used.

`main` Listing 6.78 shows the results generated with `XLC` for the `main`, `altivec`, and `vsx` FPUs of the `ppc` architecture port which check the generic floating-point operators, the VMX unit, and the newer VSX FPU (for details, see Section 5.4.1).

```
2f25dedb  xlc-xlct2s_vsx_t2s_v           total:   1.27%  19365    245  ulp:    23 (   1)  ftz: no
37babd22  xlc-xlct2d_vsx_t2d_v           total:   0.97%  19776    191  ulp:    27 (   1)  ftz: no
5cac5235  xlc-xlct2s_main_t2s_v          total:  22.24%  20191   4490  ulp:   667 (   1)  ftz: no
7badfacb  xlc-xlct2d_vsx_t2d_x           total:   0.16%  19776     31  ulp:    27 (   1)  ftz: no
7ccb6c0f  xlc-xlct2s_main_t2s_x          total:   3.65%  20191    737  ulp:   667 (   1)  ftz: no
7fe4ff3a  xlc-xlct2s_vsx_t2s_x           total:   0.14%  19365     27  ulp:    23 (   1)  ftz: no
83538bff  xlc-xlct2d_main_t2d_v          total:  21.15%  21221   4489  ulp:     0 (   0)  ftz: no
8ae3677c  xlc-xlct2d_main_t2d_x          total:   0.34%  21221     72  ulp:     0 (   0)  ftz: no
8f092be2  xlc-xlct2s_altivec_t2s_v       total:  12.28%  19318   2372  ulp:   865 (   1)  ftz: [???]
eb97ab2b  xlc-xlct2d_altivec_t2d_v       total:   0.00%      0      0  ulp:     0 (   0)  ftz: no
eb97ab2b  xlc-xlct2d_altivec_t2d_x       total:   0.00%      0      0  ulp:     0 (   0)  ftz: no
ec214bd6  xlc-xlct2s_altivec_t2s_x       total:   7.35%  19318   1420  ulp:   865 (   1)  ftz: [???]
```

Listing 6.78: *Summary of testing results on POWER8 with `XLC`.*

We start with the analysis of the `main` FPU in order to show `XLC` results for conversion between binary and decimal representation. When comparing the error rates of the `t2s` and `t2d` testsets for the `ppc main` FPU with those generated by `gcc`, `clang`, and `icc` for the `x86 main` FPU (see Listing 6.38)[3], they are roughly of the same magnitude for double precision, but about 3% higher for both `-vio` and `-vix` modes. Listings 6.79 and 6.80 show the corresponding `basic` evaluation function output.

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
rem       100.00%   1364/1364
sqrt       62.61%   1117/1784
b2d         1.63%   8/492
```

[3]Note that the `ppc main` and `x86 main` FPUs are identical to the `default main` FPU. Additionally, the conversions between binary and decimal representation are implemented in software, with the consequence that they do not depend on the underlying architecture, but on the compiler (or the floating-point library that the compiler uses to implement these conversions).

```
d2b         1.60%   54/3372
i         100.00%   350/350
ri         89.13%   164/184
rI        100.00%   47/47
ru         96.30%   26/27
rU        100.00%   28/28
ci        100.00%   56/56
cI        100.00%   56/56
cu        100.00%   48/48
cU        100.00%   48/48
RESULT     77.76%   15701/20191
```

Listing 6.79: *`basic` evaluation function results on POWER8 in `-vio` mode with the `t3s` testset.*

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
rem       100.00%   1364/1364
sqrt       62.61%   1117/1784
b2d        86.99%   428/492
d2b        99.82%   3366/3372
i         100.00%   350/350
ri        100.00%   184/184
rI        100.00%   47/47
ru        100.00%   27/27
rU        100.00%   28/28
ci        100.00%   56/56
cI        100.00%   56/56
cu        100.00%   48/48
cU        100.00%   48/48
RESULT     96.35%   19454/20191
```

Listing 6.80: *`basic` evaluation function results on POWER8 in `-vix` mode with the `t3s` testset.*

These results reveal that the (additional) errors stem from the square root operator which is used on this POWER8 platform by the `XLC` compiler. With Listing 6.81, we can conclude that these errors are caused by incorrect rounding (significand being 1 ulp off), and Listing 6.82 offers a probable explanation: It seems that roundTowardZero is the only rounding mode supported in the square root implementation.

```
sqrt
    em          1      1
    m           1    666
        e - exponent different
        m - mantissa different
```

Listing 6.81: *`operation_report` evaluation function results for POWER8 in `-vix` mode with the `t3s` testset.*

```
(Success rates shown)

n          93.40%   4885/5230
z         100.00%   5046/5046
u          92.63%   4590/4955
d          99.46%   4933/4960
RESULT     96.35%   19454/20191
```

Listing 6.82: *`roundings` evaluation function results for POWER8 in `-vix` mode with the `t3s` testset.*

VMX The VMX FPU (i. e. the `altivec` FPU) exhibits a substantial number of exception errors; furthermore, it reveals deficiencies in the multiplication, division, square root, and converting to integer operators (see Listing 6.83). Actually, the errors in division and square root are to be expected since the `altivec` implementation in the `ppc` architecture port makes use of the reciprocal estimates implemented in VMX and a Newton-Raphson iterative approach.

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul        98.41%   5322/5408
div        58.10%   1251/2153
sqrt       77.30%   1379/1784
fma       100.00%   4534/4534
i         100.00%   350/350
ri         85.33%   157/184
ru        100.00%   27/27
ci        100.00%   56/56
cu        100.00%   48/48
RESULT     92.65%   17898/19318
```

Listing 6.83: *`basic` evaluation function results for the `altivec` FPU on POWER8 in `-vix` mode with the `t3s` testset.*

The cause of the errors in the other two operators can be reasoned from the `operation_report` report excerpt in Listing 6.84: The sign errors in the multiplication operator stem from the multiplication operator being implemented via the VMX `fma` operator since VMX does not feature a separate multiplication operator. Furthermore, the errors in the conversion to integer operator are once again caused by only roundTowardZero being supported.

```
mul
    s          0     86
        s - Different sign
ri
    m          1     27
        m - mantissa different
```

Listing 6.84: *`operation_report` evaluation function excerpt for the `altivec` FPU on POWER8 in `-vix` mode with the `t3s` testset.*

VSX We conclude the XLC testing result analysis with a short look at VSX results which are mostly free of errors. Listings 6.85 and 6.86 show two familiar sources of errors: missing inexact exception flags in the rounding to integral operator and only roundTowardZero being supported for conversion to integers (the last conclusion being strengthened by Listing 6.87).

```
(Success rates shown)

add       100.00%   3152/3152
sub       100.00%   1622/1622
mul       100.00%   5408/5408
div       100.00%   2153/2153
sqrt      100.00%   1784/1784
fma       100.00%   4534/4534
i          37.71%   132/350
ri         85.33%   157/184
rI        100.00%   47/47
ru        100.00%   27/27
ci        100.00%   56/56
cu        100.00%   48/48
RESULT     98.73%   19120/19365
```

Listing 6.85: *`basic` evaluation function results for the `vsx` FPU on POWER8 in `-vio` mode with the `t3s` testset.*

```
(Errors, ulps, error count shown)

i
    j          0    218
        j - inexact flag not returned
ri
    m          1     27
        m - mantissa different
```

Listing 6.86: *`operation_report` evaluation function output for the `vsx` FPU on POWER8 in `-vio` mode with the `t3s` testset.*

```
(Success rates shown)

n          99.84%   5024/5032
z         100.00%   4814/4814
u          99.79%   4747/4757
d          99.81%   4753/4762
RESULT     99.86%   19338/19365
```

Listing 6.87: *`]roundings` evaluation function results for the `vsx` FPU on POWER8 in `-vix` mode with the `t3s` testset.*

VSX (gcc) We conclude the IEEE-conformity analysis of the POWER8 platform with some results generated by `gcc` 5.4 for the `vsx` FPU. We evaluate these results since `gcc` uses a set of intrinsics which is different to those of `XLC`. Listing 6.88 shows the testing summary.

```
1170a518  gcc-vsx-gcct2s_vsx_t2s_v        total:   1.91%  19318    369  ulp:    29 (   1)  ftz: no
909eb30f  gcc-vsx-gcct2d_vsx_t2d_x        total:   0.00%  19120      0  ulp:     0 (   0)  ftz: no
933265fa  gcc-vsx-gcct2d_vsx_t2d_v        total:   0.84%  19120    160  ulp:     0 (   0)  ftz: no
d03a6863  gcc-vsx-gcct2s_vsx_t2s_x        total:   0.17%  19318     33  ulp:    29 (   1)  ftz: no
```

Listing 6.88: *Summary of testing results on POWER8 with* `gcc`*.*

The errors in the double precision results are caused by missing inexact flags in the rounding to integral value operator (output omitted here). In single precision, there also were a number of inexact flags missing in all operations converting to or from integer values (include rounding to integral value), with additionally a few results rounded incorrectly as shown in Listing 6.89.

```
(Errors, ulps, error count shown)

ci
    jm         1      3
        j - inexact flag not returned
        m - mantissa different
cu
    jm         1      3
        j - inexact flag not returned
        m - mantissa different
ri
    jm         1     27
        j - inexact flag not returned
        m - mantissa different
```

Listing 6.89: `operation_report` *evaluation function output for the* `vsx` *FPU on POWER8 in* `-vio` *mode with the* `t3s` *testset (*`gcc`*).*

Overall, we can conclude from these results that the VSX unit should be preferred to VMX/AltiVec, at least on POWER8.

6.5.2 Cell

PS3, QPACE

The design of the Cell microarchitecture is interesting in that it combines a traditional CPU core (the PPU) with up to eight coprocessor elements specifically designed to speed up floating-point computations (the SPUs). For a brief overview of the architecture, see Section 5.4.2. The differences between the original Cell processor (such as 7 vs. 8 usable SPUs and the execution speed ratio for single vs. double precision floating-point calculations 1 : 8 or 1 : 2) and the PowerXCell 8i should not be relevant from a floating-point accuracy point of view. We used the following two user environments to generate Cell results:

- A Sony PS 3 running Fedora Core 9 (PPC64) with Linux kernel version `2.6.32`.
- A QPACE node card (cf. e. g. [Bai+09; WIK17x]) running Fedora Core 9 (PPC64) with a custom Linux kernel version `2.6.29.6-qpace`.

On the PS 3, the executables were compiled with the Cell SDK versions 3.1 and 3.2 (i. e. `ppu-gcc` and `spu-gcc` versions 4.1 and 4.3), whereas on QPACE, only the latter was used.

As discussed in Section 5.4.2, the floating-point capabilities of the FPU in the SPUs are limited, further aggravated by the fact that the `IeeeCC754++` implementation lacks proper rounding and exception support. We therefore show only results generated with the `t3sz` and `t3dn` testsets in `-vix` mode, i. e. we only test single precision operations with the roundTowardZero mode (the only rounding mode supported in single precision) and double precision with roundTiesToEven (the default mode for double precision). Additionally, we used the `IeeeCC754++` command line option `--skipsubnormal` to avoid error messages due to the SPU using FTZ and DAZ. Listings 6.90 and 6.91 show the output of the `basic` evaluation function for corresponding test runs on a QPACE node.

```
(Success rates shown)

add         94.12%   528/561
sub         93.08%   269/289
mul         70.71%   565/799
div         34.80%   119/342
sqrt        76.36%   281/368
fma         81.33%   601/739
ri         100.00%   44/44
ru         100.00%   26/26
ci         100.00%   14/14
cu         100.00%   12/12
RESULT      76.99%   2459/3194

   GROUPED

basic       74.38%   1481/1991
extra       79.67%   882/1107
conv       100.00%   96/96
```

Listing 6.90: *`basic` evaluation function results on QPACE in `-vix` mode with the `t3sz` testset.*

As can be seen from Listing 6.91, double precision operations are indeed IEEE-conforming in roundTiesToEven mode when ignoring test vectors which contain subnormals. Due to the limitations in the `spu` FPU implementation, no conclusions can be drawn for the other rounding modes or with regard to exception flags.

For operands in single precision, the results are worse (as expected). Listing 6.92 shows an excerpt of the (rather long) `error_report` evaluation function from which the two most relevant error sources can be concluded: handling of infinities and NaNs. This behaviour is in line with the peculiar choice of using the maximum exponent for normalised floating-point numbers instead of representing infinities and NaNs as required by IEEE 754-2008.

```
(Success rates shown)

add       100.00%   561/561
sub       100.00%   289/289
mul       100.00%   793/793
fma       100.00%   767/767
rt        100.00%   43/43
RESULT    100.00%   2453/2453

   GROUPED

basic     100.00%   1643/1643
extra     100.00%   767/767
conv      100.00%   43/43
```

Listing 6.91: *`basic` evaluation function results on QPACE in `-vix` mode with the `t3dn` testset.*

```
eh      - exponent different
        - result is not an infinity
    add        0      4
    div        0      5
    mul        0      6
    sub        0      2
eha     - fma error
        - exponent different
        - result is not an infinity
    fma        0      5
em      - exponent different
        - mantissa different
    div        1     67
    sqrt       1      6
emh     - exponent different
        - result is not an infinity
        - mantissa different
    div        0      8
    mul        0      6
emha    - fma error
        - exponent different
        - result is not an infinity
        - mantissa different
    fma        0      3
mg      - result is NaN, expected infinity
        - mantissa different
    add        0      6
    div        0     12
    mul        0     34
    sub        0      4
mga     - fma error
        - result is NaN, expected infinity
        - mantissa different
    fma        0     22
n       - result is not a NaN
    add        0      4
    div        0     13
    fma        0      4
    mul        0      4
```

```
    sub        0      2
pn      - result is not a NaN
        - invalid flag not returned
    add        0      2
    div        0      8
    fma        0      5
    mul        0      8
    sqrt       0     15
    sub        0      1
```

Listing 6.92: *`error_report` evaluation function excerpt on QPACE in `-vix` mode with the `t3sn` testset.*

The results shown here were generated in both user environments, i. e. on a QPACE node as well as on the PS 3. All output files (for the respective testsets) were identical. Overall, with these small (and limited) tests, we could affirm that the Cell SPU implements floating-point operations as described in [Mue+05] and that double precision computations are (mostly) IEEE-conforming.

6.6 Accelerators

In this section, we examine the IEEE-conformity of GPGPUs representative for the different variants of accelerators. The tests were executed on a compute server with an Intel Xeon E5-2699 v4 CPU (Broadwell microarchitecture, cf. [INT16d]) running openSUSE Leap 42.3 with Linux kernel `4.4.87-25-default`. This platform serves as a host system for two NVidia Quadro P6000 GPUs (see [NVi16b]) based on Pascal microarchitecture, cf. [NVi16a]. `gcc` 4.8.5 and the CUDA toolkit 8.0 were used to compile the `IeeeCC754++` executables.

We chose this system as the user environment for GPGPU testing since it contains a recent GPGPU and allows for direct comparison of results generated with two different APIs, namely CUDA and OpenCL.

6.6.1 CUDA

We begin the analysis of the P6000 GPUs with the more "native" CUDA API. The results displayed in Listing 6.93 were generated with the CUDA toolkit 8.0 and compute capability version 6.1. Since the GPU only supports the roundTiesToEven rounding mode for most operations (with the exception of the `cudai` FPU, cf. Section 5.5.1), the testsets `t3s`, `t3d`, `t3sn`, and `t3dn` were used.

```
014ce2cb  nv-nvcc80t3s_cuda_t3s_v        total:  50.64%  20861  10563  ulp:  3010 (   1)  ftz: [???]
014ce2cb  nv-nvcc80t3s_main_t3s_v        total:  50.64%  20861  10563  ulp:  3010 (   1)  ftz: [???]
045e8731  nv-nvcc80t3d_cuda_t3d_x        total:  18.24%  21312   3887  ulp:  3093 (   1)  ftz: [???]
045e8731  nv-nvcc80t3d_main_t3d_x        total:  18.24%  21312   3887  ulp:  3093 (   1)  ftz: [???]
259b0920  nv-nvcc80t3dn_cudai_t3dn_x     total:   0.00%   4656      0  ulp:     0 (   0)  ftz: no
25e0fc58  nv-nvcc80t3d_cudai_t3d_x       total:   0.00%  17896      0  ulp:     0 (   0)  ftz: no
264b67bc  nv-nvcc80t3sn_cudai_t3sn_x     total:   0.00%   4563      0  ulp:     0 (   0)  ftz: no
3ff3a2ee  nv-nvcc80t3sn_cuda_t3sn_x      total:   0.00%   5399      0  ulp:     0 (   0)  ftz: no
3ff3a2ee  nv-nvcc80t3sn_main_t3sn_x      total:   0.00%   5399      0  ulp:     0 (   0)  ftz: no
3ff3a2ee  nv-nvcc80t3sn_cuda_rn_t3sn_x   total:   0.00%   5399      0  ulp:     0 (   0)  ftz: no
```

```
46d3f366  nv-nvcc80t3s_cuda_rn_t3s_x      total:   0.00%   5399      0  ulp:     0 (   0)  ftz: no
4c42215e  nv-nvcc80t3s_cudai_t3s_v        total:  53.70%  17525   9411  ulp:     0 (   0)  ftz: no
5c39e730  nv-nvcc80t3d_cuda_rn_t3d_v      total:  52.50%   5512   2894  ulp:     0 (   0)  ftz: no
6d60a76d  nv-nvcc80t3sn_cudai_t3sn_v      total:  55.77%   4563   2545  ulp:     0 (   0)  ftz: no
732fd8cb  nv-nvcc80t3dn_cuda_t3dn_v       total:  52.50%   5512   2894  ulp:     0 (   0)  ftz: no
732fd8cb  nv-nvcc80t3dn_main_t3dn_v       total:  52.50%   5512   2894  ulp:     0 (   0)  ftz: no
732fd8cb  nv-nvcc80t3dn_cuda_rn_t3dn_v    total:  52.50%   5512   2894  ulp:     0 (   0)  ftz: no
7d00aa67  nv-nvcc80t3s_cuda_t3s_x         total:  18.12%  20861   3780  ulp:  3010 (   1)  ftz: [???]
7d00aa67  nv-nvcc80t3s_main_t3s_x         total:  18.12%  20861   3780  ulp:  3010 (   1)  ftz: [???]
94d70514  nv-nvcc80t3s_cudai_t3s_x        total:   0.00%  17525      0  ulp:     0 (   0)  ftz: no
9c576abc  nv-nvcc80t3s_cuda_rn_t3s_v      total:  52.29%   5399   2823  ulp:     0 (   0)  ftz: no
a90da235  nv-nvcc80t3d_cuda_t3d_v         total:  50.89%  21312  10846  ulp:  3093 (   1)  ftz: [???]
a90da235  nv-nvcc80t3d_main_t3d_v         total:  50.89%  21312  10846  ulp:  3093 (   1)  ftz: [???]
b068f86d  nv-nvcc80t3dn_cudai_t3dn_v      total:  56.19%   4656   2616  ulp:     0 (   0)  ftz: no
b2210808  nv-nvcc80t3d_cuda_rn_t3d_x      total:   0.00%   5512      0  ulp:     0 (   0)  ftz: no
bcc381f8  nv-nvcc80t3sn_cuda_t3sn_v       total:  52.29%   5399   2823  ulp:     0 (   0)  ftz: no
bcc381f8  nv-nvcc80t3sn_main_t3sn_v       total:  52.29%   5399   2823  ulp:     0 (   0)  ftz: no
bcc381f8  nv-nvcc80t3sn_cuda_rn_t3sn_v    total:  52.29%   5399   2823  ulp:     0 (   0)  ftz: no
bf173e3d  nv-nvcc80t3d_cudai_t3d_v        total:  54.17%  17896   9694  ulp:     0 (   0)  ftz: no
ebde191e  nv-nvcc80t3dn_cuda_t3dn_x       total:   0.00%   5512      0  ulp:     0 (   0)  ftz: no
ebde191e  nv-nvcc80t3dn_main_t3dn_x       total:   0.00%   5512      0  ulp:     0 (   0)  ftz: no
ebde191e  nv-nvcc80t3dn_cuda_rn_t3dn_x    total:   0.00%   5512      0  ulp:     0 (   0)  ftz: no
```

Listing 6.93: *CUDA testing summary.*

As a first observation, results generated with the `main` and `cuda` FPUs are identical, coming at no surprise considering that the only difference between these FPUs is the use of scalar operands (`main`) and SIMD vectors (`cuda`). This also means that no vector errors occurred, i. e. all execution units inside the P6000 GPU yielded identical results. Second, no errors were found in all results generated with roundTiesToEven (in `-vix` mode), i. e. results generated with the `cuda_rn` FPU or with the `t3sn` and `t3dn` testsets. Furthermore, all results from testing the `cudai` FPU (which uses intrinsics to provide correctly rounded operators) in `-vix` mode exhibit no errors. Since CUDA (and the GPU) do not support floating-point exceptions, the corresponding `-vio` results can be ignored, and we can conclude that this user environment (with CUDA) is IEEE-conforming (when ignoring floating-point exceptions and keeping in mind that only roundTiesToEven is supported unless the corresponding intrinsics are used).

Note that results for testing the elementary functions available in CUDA can be found in Section 6.8.

6.6.2 OpenCL

For a fair comparison with the CUDA results, we used the OpenCL libraries contained in the CUDA toolkit 8.0 in the OpenCL IEEE-conformity testing. Listing 6.94 shows the corresponding summary of the results generated with `gcc` 4.8.5 and the `t3s`, `t3d`, `t3sn`, and `t3dn` testsets, with the `-vio` mode results omitted (since OpenCL also does not support floating-point exceptions).

```
1ff0ebae  cl-gcct3d_opencl_rn_t3d_x       total:   0.16%   5512      9  ulp:     8 (   1)  ftz: no
2b1954b3  cl-gcct3dn_opencl_round_t3dn_x  total:   0.00%    181      0  ulp:     0 (   0)  ftz: no
4b54904c  cl-gcct3sn_main_t3sn_x          total:   0.59%   5399     32  ulp:    29 (   1)  ftz: [???]
4b54904c  cl-gcct3sn_opencl_t3sn_x        total:   0.59%   5399     32  ulp:    29 (   1)  ftz: [???]
4b54904c  cl-gcct3sn_opencl_rn_t3sn_x     total:   0.59%   5399     32  ulp:    29 (   1)  ftz: [???]
56f7e924  cl-gcct3d_opencl_round_t3d_x    total:   0.00%    828      0  ulp:     0 (   0)  ftz: [no ]
63ebb51a  cl-gcct3sn_opencl_round_t3sn_x  total:   0.00%     98      0  ulp:     0 (   0)  ftz: no
a5191579  cl-gcct3s_opencl_rn_t3s_x       total:   0.59%   5399     32  ulp:    29 (   1)  ftz: [???]
b0a38a5f  cl-gcct3s_main_t3s_x            total:  18.21%  20861   3798  ulp:  3023 (   1)  ftz: [???]
```

```
b0a38a5f  cl-gcct3s_opencl_t3s_x          total:  18.21%  20861   3798  ulp:  3023 (   1)  ftz: [???]
b138ebc1  cl-gcct3s_opencl_round_t3s_x    total:   0.00%    494      0  ulp:     0 (   0)  ftz: [no ]
b36b0e5c  cl-gcct3d_main_t3d_x            total:  18.20%  21312   3879  ulp:  3086 (   1)  ftz: [???]
b36b0e5c  cl-gcct3d_opencl_t3d_x          total:  18.20%  21312   3879  ulp:  3086 (   1)  ftz: [???]
c104787e  cl-gcct3dn_main_t3dn_x          total:   0.16%   5512      9  ulp:     8 (   1)  ftz: no
c104787e  cl-gcct3dn_opencl_t3dn_x        total:   0.16%   5512      9  ulp:     8 (   1)  ftz: no
c104787e  cl-gcct3dn_opencl_rn_t3dn_x     total:   0.16%   5512      9  ulp:     8 (   1)  ftz: no
```

Listing 6.94: *OpenCL testing summary, `-vix` mode.*

The first striking difference to the CUDA results is that the only tests without errors are those generated with the `opencl_round` FPU which implements the operators for which OpenCL provides rounded versions, namely conversions between floating-point formats and between floating-point numbers and integers. Consequently, error counts for the testsets containing test vectors for all rounding modes yield a large number of errors. Overall, error counts are quite low for roundTiesToEven: 9 errors for double precision and 32 errors for single precision operands (error counts for the `t3s` and `t3d` testsets are not relevant due to only roundTiesToEven being supported and these testsets containing test vectors for all rounding modes).

Listings 6.95 and 6.96 show the output of the `operation_report` evaluation function for the `t3sn` and `t3dn` of the `opencl` FPU. All errors are related to the returned result being 1 ulp off. In the case of the division operator in single precision, two results were rounded to a normalised number instead of a subnormal (error signature `jlemft`).

```
(Errors, ulps, error count shown)

div
    jem        1      2
    jlemft     0      2
    jlm        1      1
    jm         1     19
        e - exponent different
        f - flush to zero detected
        j - inexact flag not returned
        l - underflow not returned
        m - mantissa different
        t - result is normalized number, expected tiny (underflown)
ri
    jm         1      8
        j - inexact flag not returned
        m - mantissa different
```

Listing 6.95: *`operation_report` evaluation function output for the `opencl` FPU in `-vix` mode with the `t3sn` testset.*

These results can be deemed somewhat astonishing, since APIs developed by the same manufacturer and identical underlying hardware were being used. Especially the incorrectly rounded results in the division operator are difficult to explain in this context, even after examining the output of `error_list` evaluation function output, which is shown in Listing 6.97.

```
(Errors, ulps, error count shown)

ri
    jm         1      9
        j - inexact flag not returned
        m - mantissa different
```

Listing 6.96: *`operation_report` evaluation function output for the `opencl` FPU in `-vix` mode with the `t3dn` testset.*

```
[__] l.2958 s div n 00f00003 40c00000 00280000 xu => - 00280001   |  jlm 1
[__] l.2978 s div n 00bfffff 3fc00000 007fffff xu => - 00800000   |  jlemft 0
[__] l.2979 s div n 80bfffff 3fc00000 807fffff xu => - 80800000   |  jlemft 0
[__] l.3024 s div n 3fbfffff 3f7ffffe 3fc00001 x => - 3fc00000   |  jm 1
[__] l.3025 s div n bfbfffff bf7ffffe 3fc00001 x => - 3fc00000   |  jm 1
[__] l.3026 s div n bfbfffff 3f7ffffe bfc00001 x => - bfc00000   |  jm 1
[__] l.3027 s div n 3fbfffff bf7ffffe bfc00001 x => - bfc00000   |  jm 1
[__] l.3047 s div n 3f7fffff 3f7ffffe 3f800001 x => - 3f800000   |  jm 1
[__] l.3049 s div n 3f7fffff 3f7ffffc 3f800002 x => - 3f800001   |  jm 1
[__] l.3050 s div n 3f7ffffd 3f7ffffc 3f800001 x => - 3f800000   |  jm 1
[__] l.3066 s div n 3fc00001 3f800001 3fc00000 x => - 3fbfffff   |  jm 1
[__] l.3067 s div n bfc00001 bf800001 3fc00000 x => - 3fbfffff   |  jm 1
[__] l.3068 s div n bfc00001 3f800001 bfc00000 x => - bfbfffff   |  jm 1
[__] l.3069 s div n 3fc00001 bf800001 bfc00000 x => - bfbfffff   |  jm 1
[__] l.3078 s div n 3f7ffffe 3f7fffff 3f7fffff x => - 3f800000   |  jem 1
[__] l.3079 s div n 3f7ffffd 3f7fffff 3f7ffffe x => - 3f7fffff   |  jm 1
[__] l.3081 s div n 3f7ffffc 3f7fffff 3f7ffffd x => - 3f7ffffe   |  jm 1
[__] l.3083 s div n 3f7ffffc 3f7ffffd 3f7fffff x => - 3f800000   |  jem 1
[__] l.3084 s div n 3f7fffff8 3f7ffffd 3f7fffffb x => - 3f7ffffc   |  jm 1
[__] l.3087 s div n 3f7fffff7 3f7ffffb 3f7ffffc x => - 3f7ffffd   |  jm 1
[__] l.3088 s div n 007ffffe 00ffffff 3effffffd x => - 3effffffe   |  jm 1
[__] l.3089 s div n 807ffffe 80ffffff 3effffffd x => - 3effffffe   |  jm 1
[__] l.3090 s div n 807ffffe 00ffffff beffffffd x => - beffffffe   |  jm 1
[__] l.3091 s div n 007ffffe 80ffffff beffffffd x => - beffffffe   |  jm 1
[__] l.5426 s ri n 3f7fffff 00000000 00000001 x => - 00000000   |  jm 1
[__] l.5427 s ri n bf7fffff 00000000 ffffffff x => - 00000000   |  jm 0
[__] l.5428 s ri n 3fffffff 00000000 00000002 x => - 00000001   |  jm 1
[__] l.5429 s ri n bfffffff 00000000 fffffffe x => - ffffffff   |  jm 1
[__] l.5430 s ri n 3fc00000 00000000 00000002 x => - 00000001   |  jm 1
[__] l.5431 s ri n bfc00000 00000000 fffffffe x => - ffffffff   |  jm 1
[__] l.5434 s ri n 3fc00001 00000000 00000002 x => - 00000001   |  jm 1
[__] l.5435 s ri n bfc00001 00000000 fffffffe x => - ffffffff   |  jm 1

  Found 32 errors.
```

Listing 6.97: *`error_list` evaluation function output for the `opencl` FPU in `-vix` mode with the `t3sn` testset.*

6.7 Software

After having examined different user environments based on a range of hardware platforms, this section covers software based floating-point environments. In par-

ticular, we take a look at two software floating-point libraries targeted at different use cases (SoftFloat, which provides a software floating-point implementation for the five precisions covered in this thesis, and MPFR, which provides an arbitrary precision floating-point library) and at a virtual machine based floating-point environment (`Java`).

All results presented in this section were generated on a notebook featuring an Intel Core i7-4800QM mobile processor (Haswell microarchitecture, cf. [INT13b]) with openSUSE Leap 42.2 and Linux kernel `4.4.92-18.36-default`. The pre-installed `gcc` 4.8.5 was used to compile all test executables.

6.7.1 SoftFloat

The `softfloat` architecture port implements the basic arithmetic operators and all conversions between floating-point and integer formats for half, single, and double precision. All five rounding modes of IEEE 754-2008 are supported. Listing 6.98 shows a summary of the testing results generated in the user environment described above for SoftFloat Release 3d with the testsets `t3s` and `t3d`.

```
41ade70d  softfloat-gcct3s_main_t3s_x     total:   0.00%  25819      0  ulp:     0 (   0)  ftz: no
6e0ab3f9  softfloat-gcct3d_main_t3d_x     total:   0.00%  26382      0  ulp:     0 (   0)  ftz: no
c6fd86ed  softfloat-gcct3s_main_t3s_v     total:   0.00%  25819      0  ulp:     0 (   0)  ftz: no
f9fa1886  softfloat-gcct3d_main_t3d_v     total:   0.00%  26382      0  ulp:     0 (   0)  ftz: no
```

Listing 6.98: *SoftFloat testing summary.*

For single and double precision operands, no errors were found for all executed test vectors. Listings 6.99 and 6.100 show the operands that were tested and that in addition to the four classic rounding modes, no errors were found for roundTiesToAway.

```
(Success rates shown)

add       100.00%   3937/3937
sub       100.00%   2026/2026
mul       100.00%   6765/6765
div       100.00%   2693/2693
rem       100.00%   1705/1705
sqrt      100.00%   2432/2432
fma       100.00%   5815/5815
ct        100.00%   100/100
rt        100.00%   337/337
ri        100.00%   230/230
rI        100.00%   47/47
ru        100.00%   27/27
rU        100.00%   28/28
ci        100.00%   60/60
cI        100.00%   70/70
cu        100.00%   50/50
cU        100.00%   60/60
RESULT    100.00%   26382/26382
```

Listing 6.99: *`basic` evaluation function results for SoftFloat with the `t3d` testset.*

```
(Success rates shown)

n          100.00%   5421/5421
z          100.00%   5236/5236
u          100.00%   5149/5149
d          100.00%   5156/5156
a          100.00%   5420/5420
```

Listing 6.100: *`roundings` evaluation function results for SoftFloat with the `t3d` testset.*

`IeeeCC754++` supports the half precision format, but the test vectors contained in the testsets are preliminary as discussed in Section 3.1.11. Nonetheless, we show experimental results of executing SoftFloat test runs with the `t3h` testset in Listing 6.101. Additionally, Listing 6.102 provides the `operation_report` output for these results and Listing 6.103 the corresponding `roundings` results.

```
0c4a778a  softfloat-gcct3h_main_t3h_x     total:   2.31%  24467    565  ulp:    84 (   6)  ftz: no
63f8f9e7  softfloat-gcct3h_main_t3h_v     total:   2.31%  24467    565  ulp:    84 (   6)  ftz: no
```

Listing 6.101: *SoftFloat testing summary, half precision.*

```
div
    jkeh       0      3
    jkem       0      2
    s          0     30
        e - exponent different
        h - result is not an infinity
        j - inexact flag not returned
        k - overflow flag not returned
        m - mantissa different
        s - Different sign
sqrt
    ijsem      0     20
    ise        0      5
    jem        0     20
    m          0    480
    xem        0      5
        e - exponent different
        i - invalid not expected
        j - inexact flag not returned
        m - mantissa different
        s - Different sign
        x - inexact not expected
```

Listing 6.102: *`operation_report` evaluation function results for SoftFloat with the `t3h` testset.*

```
(Success rates shown)

n          96.79%   4861/5022
z          98.31%   4713/4794
u          98.32%   4732/4813
d          98.32%   4735/4816
```

```
a           96.79%   4861/5022
RESULT      97.69%   23902/24467
```

Listing 6.103: *`roundings` evaluation function results for SoftFloat with the `t3h` testset.*

Overall, most test vectors could be executed successfully (i. e. they returned no errors) for all five rounding modes. Furthermore, errors are only found in the division and square root operators, with most errors being reported for the latter. However, it is not possible to draw conclusions concerning the half precision IEEE-conformity of the SoftFloat implementation prior to a detailed analysis which of the supplied test vectors are actually valid for precisions smaller than single.

6.7.2 MPFR

Listing 6.104 shows results for the MPFR arbitrary-precision floating-point library with the testsets `t3s` and `t3d` which were generated on the same platform used for the SoftFloat results.

```
106fc1cd  mpfr-gcct3s_mpfrdef_t3s_x       total:   0.07%  20546     14  ulp:     0 (   0)  ftz: [???]
50d2656c  mpfr-gcct3s_main_t3s_v          total:  12.98%  20546   2667  ulp:     0 (   0)  ftz: no
540242d1  mpfr-gcct3d_mpfrdef_t3d_x       total:   0.07%  20933     14  ulp:     0 (   0)  ftz: [???]
621f8bb5  mpfr-gcct3d_main_t3d_v          total:  13.41%  20933   2807  ulp:     0 (   0)  ftz: no
646db5c7  mpfr-gcct3s_main_t3s_x          total:   0.00%  20546      0  ulp:     0 (   0)  ftz: no
7ad2ee50  mpfr-gcct3d_mpfrdef_t3d_v       total:   7.90%  20933   1654  ulp:     0 (   0)  ftz: [???]
85ac29fe  mpfr-gcct3s_mpfrdef_t3s_v       total:   7.39%  20546   1518  ulp:     0 (   0)  ftz: [???]
fb9a1c5d  mpfr-gcct3d_main_t3d_x          total:   0.00%  20933      0  ulp:     0 (   0)  ftz: no
```

Listing 6.104: *MPFR testing summary.*

MPFR supports the five exceptions defined in IEEE 754-2008, although they are implemented in a slightly different manner (there is an additional range error exception, and some operators signal exceptions in slightly non-conforming ways, see also [GNU16d]). As can be seen from the `-vio` mode results for the `main` FPU in comparison to the `-vix` mode results, this behaviour leads to a significant number of errors. However, when explicitly enabling subnormal support (which has been done in the `main` FPU), MPFR yields fully IEEE-conforming results in the sense that all returned floating-point numbers are correctly rounded.

In the default MPFR configuration (i. e. with subnormal support disabled), MPFR shows a different underflow behaviour due to the extended internal precision. However, a small number of returned floating-point values is incorrectly rounded as can be seen from Listings 6.105 and 6.106. The listings also reveal the reason for the errors only occurring for multiplication and division (the `fma` cases check the built-in multiplication) in roundTiesToEven mode: Due to the extended intermediate precision, the floating-point number closest to the computed result is not the largest subnormal, but the smallest normalised number (in magnitude).

```
lemft   - exponent different
        - flush to zero detected
        - underflow not returned
        - mantissa different
        - result is normalized number, expected tiny (underflown)
    div         0      2
    mul         0      8
lemfta  - fma error
        - exponent different
        - flush to zero detected
        - underflow not returned
        - mantissa different
        - result is normalized number, expected tiny (underflown)
    fma         0      4
```

Listing 6.105: *`error_report` evaluation function results for the MPFR `mpfrdef` FPU with the `t3d` testset.*

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n             0      0     14      5424
z             0      0      0      5208
u             0      0      0      5146
d             0      0      0      5155
RESULT        0      0     14     20933
```

Listing 6.106: *`ulp` evaluation function results for the MPFR `mpfrdef` FPU with the `t3d` testset.*

6.7.3 Java

We conclude this section with some results for the `java` architecture port, once again generated on the platform which was used for SoftFloat testing. For the tests, we used the `Java` SE Development Kit (JDK) versions 1.7.0_67 and 1.8.0_20. Listing 6.107 shows the corresponding testing summary for the `t2s` and `t2d` testsets (which include conversions between binary and decimal representations) in `-vix` mode (since `Java` does not support floating-point exceptions). Only roundTiesToEven is supported by the `IeeeCC754++ java` architecture.

```
2f28542f  java-java170t2d_main_t2d_x      total:   1.68%   6073    102  ulp:    61 (   1)  ftz: no
2f28542f  java-java180t2d_main_t2d_x      total:   1.68%   6073    102  ulp:    61 (   1)  ftz: no
2f28542f  java-java170t2d_strict_t2d_x    total:   1.68%   6073    102  ulp:    61 (   1)  ftz: no
2f28542f  java-java180t2d_strict_t2d_x    total:   1.68%   6073    102  ulp:    61 (   1)  ftz: no
321a0e8f  java-java170t2s_strict_t2s_x    total:   2.27%   5992    136  ulp:   104 (   1)  ftz: no
360daca6  java-java180t2s_strict_t2s_x    total:   1.22%   5992     73  ulp:    41 (   1)  ftz: no
778f3a71  java-java180t2s_main_t2s_x      total:   2.12%   5992    127  ulp:    45 (   1)  ftz: no
9a488433  java-java170t2s_main_t2s_x      total:   3.12%   5992    187  ulp:   105 (   1)  ftz: no
```

Listing 6.107: *`Java` testing summary.*

For this architecture port, significant differences between single and double precision exist. For the latter, the results generated with both `Java` versions are

identical for the `main` and `strict` FPUs; Listing 6.108 shows the reasons for these errors: some conversions to decimal format yielded incorrect decimal numbers, `fma` is not fused (which comes at no surprise since it is implemented with regular multiplications and additions in the `java` port, cf. Section 5.7.1), and finally, some test vectors converting floating-point values to integers were incorrectly rounded.

```
b2d
    jd          0     17
        d - different decimal representation
        j - inexact flag not returned
fma
    jlsa        0     23
    jma         1     29
    ma          1     24
        a - fma error
        j - inexact flag not returned
        l - underflow not returned
        m - mantissa different
        s - Different sign
ri
    jm          1      9
        j - inexact flag not returned
        m - mantissa different
```

Listing 6.108: *`operation_report` evaluation function results for `Java` with the `t3d` testset.*

For single precision, the situation is more complex: First, testing with JDK 1.8 yielded better results than testing with JDK 1.7, and second, the `strict` FPU returned fewer errors than the respective `main` FPU. Indeed, the JDK 1.8 generated `strict` FPU in single precision showed the minimum number of errors overall, whereas the `main` FPU compiled with JDK 1.7 showed the maximum number of errors. Listings 6.109 and 6.110 show corresponding `operation_report` output.

```
b2d
    jd          0      9
        d - different decimal representation
        j - inexact flag not returned
fma
    jlsa        0     23
    jma         2     14
    ma          1     24
        a - fma error
        j - inexact flag not returned
        l - underflow not returned
        m - mantissa different
        s - Different sign
ri
    jm          1      3
        j - inexact flag not returned
        m - mantissa different
```

Listing 6.109: *`operation_report` evaluation function results for the `strict` FPU in conjunction with JDK 1.8 and the `t3s` testset.*

```
b2d
    jd          0      9
        d - different decimal representation
        j - inexact flag not returned
d2b
    jm          1     60
        j - inexact flag not returned
        m - mantissa different
fma
    jlsa        0     23
    jma         2     14
    ma          1     24
        a - fma error
        j - inexact flag not returned
        l - underflow not returned
        m - mantissa different
        s - Different sign
i
    e           0      3
    eh          0      2
    em          0     11
    je          0      6
    jem         0     10
    jm          0      6
    js          0      6
    jse         0      2
    n           0      2
    s           0      1
        e - exponent different
        h - result is not an infinity
        j - inexact flag not returned
        m - mantissa different
        n - result is not a NaN
        s - Different sign
ri
    jm          1      8
        j - inexact flag not returned
        m - mantissa different
```

Listing 6.110: *`operation_report` evaluation function results for the `main` FPU in conjunction with JDK 1.7 and the `t3s` testset.*

Listing 6.109 reveals that for JDK 1.8, single and double precision operators behave similarly for the `strict` FPU, while the `main` FPU exhibits errors in the round to integral value operator similar to those shown in Listing 6.110. The latter output finally shows that with JDK 1.7, additional errors were found when converting decimal input to (binary) floating-point numbers. Overall, we can conclude that JDK 1.7 should be avoided when performing single precision floating-point calculations.

6.8 Elementary functions

`IeeeCC754++` adds support for the elementary functions recommended (but not required) by IEEE 754-2008. In Chapter 4, we described the newly added operators

as well as the test vectors that were added to check the IEEE-conformity of the operators. In this section, we discuss some results of conformity checking for the (logical) FPUs inside `IeeeCC754++` that support a subset of the elementary operators: the `c99` and `cpp11` FPUs in the `default` architecture port, the `mpfr main` FPU, the `crlibm` architecture, and the `cuda` FPU in the `nv` port. The CUDA results were generated in the user environment described in Section 6.6. For all other results, we used the mobile platform detailed in Section 6.7.

Since the extent to which `IeeeCC754++` is supported for the different resulting user environments varies, we split up the test vectors into the following testsets for the IEEE-conformity testing: `tTs`, `tQs`, `tPs`, `tTd`, `tQd`, and `tPd` which check trigonometric operators (`tT`), exponentials and logarithms (`tQ`), and power functions (`tP`) in single and double precision (for details on the labelling of test vector collections, see Section 4.7.3). Additionally, we discuss results generated with this collection of operators for double precision in roundTiesToEven mode, i. e. for the testsets `tTdn`, `tQdn`, and `tPdn`.

This section is organised as follows: We first take a brief look at the results generated with the `c99` and `cpp11` FPUs of the `default` architecture and discuss the differences found in the user environment which uses `gcc` 4.8.5 as compiler. We then examine the results for single and double precision in all rounding modes for the FPUs listed above, broken up by operation type (trigonometric operators, exponentials and logarithms, and power functions). We conclude this section by showing results for all elementary operators in double precision with the roundTiesToEven rounding mode.

6.8.1 `C99` vs. `C++11`

We start the elementary operator testing by comparing the results generated with the `C99` and `C++11` FPUs. The corresponding summaries are displayed in Listing 6.111.

```
094f2173  c99-gcctQs_c99_tQs_x          total:   0.00%    252      0  ulp:     0 (   0)  ftz: no
094f2173  cpp11-gcctQs_cpp11_tQs_x      total:   0.00%    252      0  ulp:     0 (   0)  ftz: no
0b26f944  c99-gcctTd_c99_tTd_v          total:  41.85%   1240    519  ulp:   256 (   1)  ftz: [yes]
0b26f944  cpp11-gcctTd_cpp11_tTd_v      total:  41.85%   1240    519  ulp:   256 (   1)  ftz: [yes]
1f2e0cd4  c99-gcctQdn_c99_tQdn_x        total:   5.47%    201     11  ulp:    11 (   1)  ftz: no
1f2e0cd4  cpp11-gcctQdn_cpp11_tQdn_x    total:   5.47%    201     11  ulp:    11 (   1)  ftz: no
26436bf7  c99-gcctPdn_c99_tPdn_v        total:   2.70%    853     23  ulp:     2 (   1)  ftz: no
26436bf7  cpp11-gcctPdn_cpp11_tPdn_v    total:   2.70%    853     23  ulp:     2 (   1)  ftz: no
3d4b9cc2  c99-gcctTs_c99_tTs_v          total:   0.00%    292      0  ulp:     0 (   0)  ftz: no
3d4b9cc2  cpp11-gcctTs_cpp11_tTs_v      total:   0.00%    292      0  ulp:     0 (   0)  ftz: no
464e340d  c99-gcctQs_c99_tQs_v          total:   0.00%    252      0  ulp:     0 (   0)  ftz: no
464e340d  cpp11-gcctQs_cpp11_tQs_v      total:   0.00%    252      0  ulp:     0 (   0)  ftz: no
54e2076a  c99-gcctTdn_c99_tTdn_x        total:   2.58%    310      8  ulp:     8 (   1)  ftz: no
54e2076a  cpp11-gcctTdn_cpp11_tTdn_x    total:   2.58%    310      8  ulp:     8 (   1)  ftz: no
5865b65c  c99-gcctTdn_c99_tTdn_v        total:  30.65%    310     95  ulp:     8 (   1)  ftz: no
5865b65c  cpp11-gcctTdn_cpp11_tTdn_v    total:  30.65%    310     95  ulp:     8 (   1)  ftz: no
6023c369  c99-gcctQdn_c99_tQdn_v        total:  11.94%    201     24  ulp:    11 (   1)  ftz: no
6023c369  cpp11-gcctQdn_cpp11_tQdn_v    total:  11.94%    201     24  ulp:    11 (   1)  ftz: no
79b744bb  c99-gcctPs_c99_tPs_v          total:   1.54%   2596     40  ulp:     0 (   0)  ftz: no
79b744bb  cpp11-gcctPs_cpp11_tPs_v      total:   1.54%   2596     40  ulp:     0 (   0)  ftz: no
84b7d3ed  c99-gcctPs_c99_tPs_x          total:   0.31%   2596      8  ulp:     0 (   0)  ftz: no
84b7d3ed  cpp11-gcctPs_cpp11_tPs_x      total:   0.31%   2596      8  ulp:     0 (   0)  ftz: no
93368e6d  c99-gcctTs_c99_tTs_x          total:   0.00%    292      0  ulp:     0 (   0)  ftz: no
93368e6d  cpp11-gcctTs_cpp11_tTs_x      total:   0.00%    292      0  ulp:     0 (   0)  ftz: no
```

```
a1ce669c  c99-gcctQd_c99_tQd_v           total:  18.19%    797    145  ulp:   101 (   1)  ftz: [???]
a1ce669c  cpp11-gcctQd_cpp11_tQd_v       total:  18.19%    797    145  ulp:   101 (   1)  ftz: [???]
c175209f  c99-gcctPd_c99_tPd_v           total:  29.38%   2604    765  ulp:   695 (   1)  ftz: no
c175209f  cpp11-gcctPd_cpp11_tPd_v       total:  29.38%   2604    765  ulp:   695 (   1)  ftz: no
c67345e3  c99-gcctQd_c99_tQd_x           total:  13.17%    797    105  ulp:   101 (   1)  ftz: [???]
c67345e3  cpp11-gcctQd_cpp11_tQd_x       total:  13.17%    797    105  ulp:   101 (   1)  ftz: [???]
c8ae3e34  c99-gcctTd_c99_tTd_x           total:  21.85%   1240    271  ulp:   256 (   1)  ftz: [yes]
c8ae3e34  cpp11-gcctTd_cpp11_tTd_x       total:  21.85%   1240    271  ulp:   256 (   1)  ftz: [yes]
f1dc3671  c99-gcctPdn_c99_tPdn_x         total:   0.47%    853      4  ulp:     2 (   1)  ftz: no
f1dc3671  cpp11-gcctPdn_cpp11_tPdn_x     total:   0.47%    853      4  ulp:     2 (   1)  ftz: no
fb03ca2d  c99-gcctPd_c99_tPd_x           total:  27.00%   2604    703  ulp:   695 (   1)  ftz: no
fb03ca2d  cpp11-gcctPd_cpp11_tPd_x       total:  27.00%   2604    703  ulp:   695 (   1)  ftz: no
```

Listing 6.111: *Testing summary for elementary operators with `C99` and `C++11`.*

Since the mathematical library included in most `C++11` compilers makes use of the mathematical functions standardised in `C99` (in fact, the same set of functions has been standardised in `C++17`, see also Section 5.1.1) and the above results were generated with the same compiler, we can expect the testing results to be identical. Listing 6.111 confirms this expectation: For all variations of precision, testset, and testing mode, the logfiles generated by `IeeeCC754++` with the `c99` and `cpp11` FPUs of the `default` architecture port are identical as shown by the corresponding checksums being equal. Therefore, we only discuss the `C99` results in the next sections and omit the `C++11` results.

6.8.2 Trigonometric operators

Listings 6.112 and 6.113 show summaries of testing the trigonometric operators with single and double precision operands, respectively.

```
3d4b9cc2  c99-gcctTs_c99_tTs_v           total:   0.00%    292      0  ulp:     0 (   0)  ftz: no
93368e6d  c99-gcctTs_c99_tTs_x           total:   0.00%    292      0  ulp:     0 (   0)  ftz: no
6cb80b89  mpfr-gcctTs_main_tTs_v         total:   2.74%    292      8  ulp:     0 (   0)  ftz: no
8f793ba6  mpfr-gcctTs_main_tTs_x         total:   0.00%    292      0  ulp:     0 (   0)  ftz: no
e39389e4  nv-nvcc80tTs_cuda_tTs_v        total:  41.10%    292    120  ulp:     0 (   0)  ftz: no
d29ba660  nv-nvcc80tTs_cuda_tTs_x        total:   0.00%    292      0  ulp:     0 (   0)  ftz: no
```

Listing 6.112: *Testing summary for trigonometric operators, single precision.*

```
0b26f944  c99-gcctTd_c99_tTd_v           total:  41.85%   1240    519  ulp:   256 (   1)  ftz: [yes]
c8ae3e34  c99-gcctTd_c99_tTd_x           total:  21.85%   1240    271  ulp:   256 (   1)  ftz: [yes]
862b75fa  crlibm-gcctTd_main_tTd_v       total:  29.12%    728    212  ulp:     0 (   0)  ftz: no
402ffb77  crlibm-gcctTd_main_tTd_x       total:   0.00%    728      0  ulp:     0 (   0)  ftz: no
8329c974  mpfr-gcctTd_main_tTd_v         total:   0.65%   1240      8  ulp:     0 (   0)  ftz: no
d45395e9  mpfr-gcctTd_main_tTd_x         total:   0.00%   1240      0  ulp:     0 (   0)  ftz: no
2ec1a3a1  nv-nvcc80tTd_cuda_tTd_v        total:  84.84%   1240   1052  ulp:   301 (   1)  ftz: [yes]
eed78658  nv-nvcc80tTd_cuda_tTd_x        total:  28.15%   1240    349  ulp:   301 (   1)  ftz: [yes]
```

Listing 6.113: *Testing summary for trigonometric operators, double precision.*

In single precision, all returned floating-point numbers were correct, with a large number of exceptions being wrong in the CUDA case (since floating-point exceptions are not supported) and the "divide by zero" exception not being returned for the `atanh` operator of the MPFR implementation (see Listing 6.114). For the `C99` implementation, no errors were encountered during testing. Note that

there are no CRlibm results listed in Listing 6.112 since this library only supports double precision operands.

```
atanh
    q          0      8
        q - divide flag not returned
```

Listing 6.114: *operation_report evaluation function results for the mpfr main FPU with the tTs testset.*

For double precision, a significantly larger number of errors is reported, especially for the C99 and CUDA FPUs. This is due to the fact that for single precision, mainly simple test cases and special cases listed in IEEE 754-2008 are contained in the testsets since we did not find hard to round cases in the literature. However, for double precision, we added a large number of hard and worst to round cases as new test vectors (see also Section 4.4.1). The most striking result of double precision testing with trigonometric operators is that CRlibm and MPFR compute correctly rounded results for all test vectors, even for the just mentioned hard cases. Both FPUs do not handle exceptions in a fully IEEE-conforming way, the MPFR FPU showing the same "divide by zero" errors in the atanh operator and CRlibm not generating proper exceptions due to design considerations concerning operator performance (as detailed in the CRlibm documentation [Dar+06]).

The errors reported for the C99 and CUDA FPUs are mostly related to results not being rounded correctly, i. e. being 1 ulp off. When looking at the ulp evaluation function output for these two FPUs (Listings 6.115 and 6.116), one can see that the error rate for results in roundTiesToEven are almost negligible. We can therefore conclude that the algorithms used to compute the floating-point results are exact enough in principle, but not enough effort is invested (either in algorithm design or computing power) to return correctly rounded results for all rounding modes.

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n            8      0      0       310
z           57      0      7       310
u           88      0      4       310
d          101      2      4       310
RESULT     254      2     15      1240
```

Listing 6.115: *ulp evaluation function results for the c99 FPU with the tTd testset.*

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n           19      0      0       310
z           73      0     24       310
u          100      0      8       310
d          109      0     16       310
```

```
RESULT     301      0     48      1240
```

Listing 6.116: *`ulp` evaluation function results for the `cuda` FPU with the `tTd` testset.*

6.8.3 Exponentials and logarithms

For the exponential and logarithmic operators, the situation is similar to the trigonometric operators. The result summaries for the testsets `tQs` and `tQd` can be found in Listing 6.117 and 6.118.

```
464e340d  c99-gcctQs_c99_tQs_v           total:   0.00%    252      0  ulp:     0 (   0)  ftz: no
094f2173  c99-gcctQs_c99_tQs_x           total:   0.00%    252      0  ulp:     0 (   0)  ftz: no
88349301  mpfr-gcctQs_main_tQs_v         total:  15.94%    276     44  ulp:     0 (   0)  ftz: no
177bac8a  mpfr-gcctQs_main_tQs_x         total:   0.00%    276      0  ulp:     0 (   0)  ftz: no
c84c1dbe  nv-nvcc80tQs_cuda_tQs_v        total:  46.15%    208     96  ulp:     0 (   0)  ftz: no
9704577c  nv-nvcc80tQs_cuda_tQs_x        total:   2.88%    208      6  ulp:     0 (   0)  ftz: no
```

Listing 6.117: *Testing summary for exponentials and logarithms, single precision.*

```
a1ce669c  c99-gcctQd_c99_tQd_v           total:  18.19%    797    145  ulp:   101 (   1)  ftz: [???]
c67345e3  c99-gcctQd_c99_tQd_x           total:  13.17%    797    105  ulp:   101 (   1)  ftz: [???]
a1278b21  crlibm-gcctQd_main_tQd_v       total:  10.85%    645     70  ulp:     0 (   0)  ftz: no
f2b6b200  crlibm-gcctQd_main_tQd_x       total:   0.00%    645      0  ulp:     0 (   0)  ftz: no
050be749  mpfr-gcctQd_main_tQd_v         total:   4.72%    933     44  ulp:     0 (   0)  ftz: no
94b8d312  mpfr-gcctQd_main_tQd_x         total:   0.00%    933      0  ulp:     0 (   0)  ftz: no
bfe8b081  nv-nvcc80tQd_cuda_tQd_v        total:  79.44%    681    541  ulp:   144 (   1)  ftz: [???]
3a670c48  nv-nvcc80tQd_cuda_tQd_x        total:  26.58%    681    181  ulp:   144 (   1)  ftz: [???]
```

Listing 6.118: *Testing summary for exponentials and logarithms, double precision.*

Testing single precision operands with exponentials and logarithms revealed no errors for `C99` and MPFR; however, some errors (in the sense of incorrect floating-point numbers being returned) were returned for the CUDA FPU. Listing 6.119 shows that all these six cases of errors are due to overflow handling in the exponential operators `exp`, `exp2`, and `expm1`.

```
exp
    jkem       0      2
        e - exponent different
        j - inexact flag not returned
        k - overflow flag not returned
        m - mantissa different
exp2
    jkem       0      2
        e - exponent different
        j - inexact flag not returned
        k - overflow flag not returned
        m - mantissa different
expm1
    jkem       0      2
        e - exponent different
```

```
        j - inexact flag not returned
        k - overflow flag not returned
        m - mantissa different
```

Listing 6.119: *operation_report evaluation function results for the nv cuda FPU with the tQs testset.*

For double precision, the reasons for the errors in the C99 and MPFR FPUs are the same as for the trigonometric operators, namely rounding being 1 ulp off for some test vectors in all rounding modes and a significantly smaller number of errors in roundTiesToEven (Listings 6.120 and 6.121). The two cases of incorrect results with a deviation from the correct result of more than 8 ulps are due to underflow handling as can be seen from Listings 6.121 and 6.122.

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n           11      0      0       201
u           24      0      1       198
z           27      0      2       198
d           39      0      1       200
RESULT     101      0      4       797
```

Listing 6.120: *ulp evaluation function results for the c99 FPU with the tQd testset.*

```
(vectors with 1, 2-8, and >8 ulps difference shown)

n            9      0      2       172
u           48      0      6       169
z           44      0     15       169
d           43      0     14       171
RESULT     144      0     37       681
```

Listing 6.121: *ulp evaluation function results for the cuda FPU with the tQd testset.*

```
jlmf    - flush to zero detected
        - inexact flag not returned
        - underflow not returned
        - mantissa different
   exp        0      2
```

Listing 6.122: *error_report evaluation function excerpt for the cuda FPU with the tQdn testset.*

6.8.4 Power operators

For the power (and root) operators, the testing results exhibit different characteristics compared to the results for the other elementary functions shown in the last

sections. First, there are no CRlibm results since this library does not implement power operators. Additionally, all of the tested FPUs returned at least a few errors. Listings 6.123 and 6.124 show the testing summaries for single and double precision.

```
79b744bb  c99-gcctPs_c99_tPs_v           total:   1.54%   2596     40  ulp:     0 (   0)  ftz: no
84b7d3ed  c99-gcctPs_c99_tPs_x           total:   0.31%   2596      8  ulp:     0 (   0)  ftz: no
dad5fd4e  mpfr-gcctPs_main_tPs_v         total:   2.13%   4508     96  ulp:     0 (   0)  ftz: no
924b1c0a  mpfr-gcctPs_main_tPs_x         total:   0.44%   4508     20  ulp:     0 (   0)  ftz: no
79e1aaf5  nv-nvcc80tPs_cuda_tPs_v        total:  79.35%   2596   2060  ulp:   815 (   1)  ftz: [yes]
32992dee  nv-nvcc80tPs_cuda_tPs_x        total:  31.93%   2596    829  ulp:   815 (   1)  ftz: [yes]
```

Listing 6.123: *Testing summary for power operators, single precision.*

```
c175209f  c99-gcctPd_c99_tPd_v           total:  29.38%   2604    765  ulp:   695 (   1)  ftz: no
fb03ca2d  c99-gcctPd_c99_tPd_x           total:  27.00%   2604    703  ulp:   695 (   1)  ftz: no
c17a8013  mpfr-gcctPd_main_tPd_v         total:   2.13%   4516     96  ulp:     0 (   0)  ftz: no
7bc0d451  mpfr-gcctPd_main_tPd_x         total:   0.44%   4516     20  ulp:     0 (   0)  ftz: no
d7149f97  nv-nvcc80tPd_cuda_tPd_v        total:  79.72%   2604   2076  ulp:   796 (   1)  ftz: [yes]
e919de74  nv-nvcc80tPd_cuda_tPd_x        total:  31.26%   2604    814  ulp:   796 (   1)  ftz: [yes]
```

Listing 6.124: *Testing summary for power operators, double precision.*

From these summaries, it is striking that the number of errors reported for the CUDA FPU is substantially larger compared to the `C99` and MPFR FPUs. The `error_report` evaluation function output for single precision in `-vix` mode shown in Listing 6.125 sheds light on the reasons for these errors which are exclusively found in the `pow` function: Most of the errors stem from the floating-point result numbers being incorrectly rounded (denoted by the error signatures "`jm`" and "`jem`"). Additionally, small problems with overflow, underflow, NaN, and sign handling were found.

```
(Operations, ulps, error count shown)

jem     - exponent different
        - inexact flag not returned
        - mantissa different
    pow         2     36
jkem    - exponent different
        - inexact flag not returned
        - overflow flag not returned
        - mantissa different
    pow         0      4
jlmf    - flush to zero detected
        - inexact flag not returned
        - underflow not returned
        - mantissa different
    pow         0      2
jm      - inexact flag not returned
        - mantissa different
    pow         1    779
pn      - result is not a NaN
        - invalid flag not returned
    pow         0      4
```

```
s       - Different sign
    pow         0      4
```

Listing 6.125: *`error_report` evaluation function excerpt for the `cuda` FPU with the `tPs` testset.*

The reasons for the (small number of) single precision errors found for the MPFR FPU mainly lie in wrong signs being returned for some test vectors in the `pow`, `rootn`, and `rsqrt` operators, cf. Listing 6.126. Additionally, four errors were found related to NaN handling. For the `C99` FPU, errors were only reported for the `pow` operator. Incidentally, these are identical to the `pow` errors of the MPFR FPU shown in Listing 6.126.

```
pow
    pn          0      4
    s           0      4
        n - result is not a NaN
        p - invalid flag not returned
        s - Different sign
rootn
    s           0      8
        s - Different sign
rsqrt
    qs          0      4
        q - divide flag not returned
        s - Different sign
```

Listing 6.126: *`operation_report` evaluation function excerpt for the `mpfr main` FPU with the `tPs` testset.*

For double precision operands, the floating-point behaviour of the tested FPUs is similar to that observed in single precision. For MPFR, the `operation_report` output is identical to that in single precision. Listing 6.127 reveals that for CUDA, the results are of similar quality as the single precision results with slightly different error counts (and sometimes signatures).

```
em      - exponent different
        - mantissa different
    pow         1      4
jem     - exponent different
        - inexact flag not returned
        - mantissa different
    pow         1     15
jkem    - exponent different
        - inexact flag not returned
        - overflow flag not returned
        - mantissa different
    pow         0      4
jlmf    - flush to zero detected
        - inexact flag not returned
        - underflow not returned
        - mantissa different
    pow         0      2
jm      - inexact flag not returned
```

```
        - mantissa different
    pow         1    773
m       - mantissa different
    pow         1      4
pn      - result is not a NaN
        - invalid flag not returned
    pow         0      4
qs      - divide flag not returned
        - Different sign
    pow         0      4
s       - Different sign
    pow         0      4
```

Listing 6.127: *`error_report` evaluation function excerpt for the `cuda` FPU with the `tPd` testset.*

The biggest difference to the single precision results is found for the `C99` FPU. The corresponding `operation_report` output is shown in Listing 6.128.

```
(Errors, ulps, error count shown)

cbrt
    xem         1     12
    xm          2     10
        e - exponent different
        m - mantissa different
        x - inexact not expected
pow
    em          1      1
    m           1    672
    pn          0      4
    s           0      4
        e - exponent different
        m - mantissa different
        n - result is not a NaN
        p - invalid flag not returned
        s - Different sign
```

Listing 6.128: *`operation_report` evaluation function excerpt for the `c99` FPU with the `tPd` testset.*

A significant number of test vectors are not rounded correctly for the `pow` operator with the usual deviation of 1 ulp from the correct result. Furthermore, some test vectors were unexpectedly incorrectly rounded for the simple test cases in the `cbrt` testset (cf. Section 4.3.1). Listing 6.129 shows that almost all of the errors are encountered in the roundTowardZero and roundTowardNegative rounding modes, whereas only a small number of errors is reported for roundTowardPositive and almost none for roundTiesToEven. This behaviour (and the fact that the returned results were identical for each Coonen source test vector in all rounding modes) suggests that the `cbrt` operator is only properly implemented for roundTiesToEven, and that the correct values for roundTowardPositive coincide with the roundTiesToEven values for the power and root test vectors.

```
(Success rates shown)

n           99.53%   849/853
z           41.78%   244/584
u           96.74%   564/583
d           41.78%   244/584
RESULT      73.00%   1901/2604
```

Listing 6.129: *`roundings` evaluation function results for the `c99` FPU with the `tPd` testset.*

6.8.5 roundTiesToEven results

Since the error counts found for the elementary operators in `-vix` mode are quite large for some of the tested FPUs, we performed further tests with testsets that only contained roundTiesToEven test vectors. This approach enables analysing how accurate (read: how IEEE-conforming) the elementary operators are implemented for the default (and most commonly used) rounding mode. Listing 6.130 shows the summary for the FPUs tested above in `-vix` mode for the testsets `tTdn`, `tQdn`, and `tPdn`.

```
54e2076a  c99-gcctTdn_c99_tTdn_x          total:   2.58%    310      8  ulp:     8 (   1)  ftz: no
1f2e0cd4  c99-gcctQdn_c99_tQdn_x          total:   5.47%    201     11  ulp:    11 (   1)  ftz: no
f1dc3671  c99-gcctPdn_c99_tPdn_x          total:   0.47%    853      4  ulp:     2 (   1)  ftz: no
8c37136a  crlibm-gcctTdn_main_tTdn_x      total:   0.00%    182      0  ulp:     0 (   0)  ftz: no
4bb508bb  crlibm-gcctQdn_main_tQdn_x      total:   0.00%    163      0  ulp:     0 (   0)  ftz: no
a0f37c40  mpfr-gcctTdn_main_tTdn_x        total:   0.00%    310      0  ulp:     0 (   0)  ftz: no
3ea33ecc  mpfr-gcctQdn_main_tQdn_x        total:   0.00%    235      0  ulp:     0 (   0)  ftz: no
72bd67a1  mpfr-gcctPdn_main_tPdn_x        total:   0.33%   1533      5  ulp:     0 (   0)  ftz: no
8189c6f1  nv-nvcc80tTdn_cuda_tTdn_x       total:   6.13%    310     19  ulp:    19 (   1)  ftz: no
226e7f4d  nv-nvcc80tQdn_cuda_tQdn_x       total:   6.40%    172     11  ulp:     9 (   1)  ftz: [???]
46a403c9  nv-nvcc80tPdn_cuda_tPdn_x       total:  14.77%    853    126  ulp:   123 (   1)  ftz: no
```

Listing 6.130: *Testing summary for elementary operators for double precision operands in `-vix` mode with roundTiesToEven.*

CRlibm yielded correctly rounded floating-point results for all test vectors supplied by `IeeeCC754++` (note that we only analyse the returned binary floating-point numbers here and ignore errors related to floating-point exceptions). MPFR also produced correct results with the exception of five errors related to sign and NaN handling (see above). For the `C99` FPU, we can see that the errors found for the power operators in the last section are not found here due to only roundTowardZero being regarded. For the trigonometric, exponential, and logarithmic operators, all errors are caused by incorrectly rounded results with a deviation of 1 ulp, spread over a number of operators (cf. Listing 6.131).

```
acosh
    m           1      2
        m - mantissa different
cosh
    m           1      4
        m - mantissa different
```

```
sinh
    m          1      2
        m - mantissa different
exp2
    m          1      2
        m - mantissa different
expm1
    m          1      2
        m - mantissa different
log10
    m          1      4
        m - mantissa different
log2
    m          1      1
        m - mantissa different
logp1
    m          1      2
        m - mantissa different
```

Listing 6.131: *`operation_report` evaluation function excerpt for the `c99` FPU with the `tTdn` and `tQdn` testsets.*

The largest error counts are found for the CUDA GPU in all three testset variants. For the trigonometric operators, all errors are due to incorrect rounding, while for exponentials and logarithms, additional problems with subnormal handling were found (cf. Listings 6.132 and Listing 6.133).

```
(Operations, ulps, error count shown)

jm      - inexact flag not returned
        - mantissa different
    acos       1      2
    acosh      1      2
    asin       1      1
    atan       1      2
    cos        1      2
    cosh       1      7
    sin        1      1
    tan        1      2
```

Listing 6.132: *`error_report` evaluation function excerpt for the `cuda` FPU with the `tTdn` testset.*

```
(Operations, ulps, error count shown)

jlm     - inexact flag not returned
        - underflow not returned
        - mantissa different
    exp2       1      1
jlmf    - flush to zero detected
        - inexact flag not returned
        - underflow not returned
        - mantissa different
    exp        0      2
jm      - inexact flag not returned
        - mantissa different
```

```
    exp         1      2
    expm1       1      2
    log10       1      2
    log2        1      2
```

Listing 6.133: *`error_report` evaluation function excerpt for the `cuda` FPU with the `tQdn` testset.*

Finally, although the error rate for the `tPdn` testset is lower than that for the `tPd` testset by a factor of two, the distribution of errors is similar as shown by Listing 6.134.

```
em      - exponent different
        - mantissa different
    pow         1      1
jem     - exponent different
        - inexact flag not returned
        - mantissa different
    pow         1     13
jm      - inexact flag not returned
        - mantissa different
    pow         1    108
m       - mantissa different
    pow         1      1
pn      - result is not a NaN
        - invalid flag not returned
    pow         0      1
qs      - divide flag not returned
        - Different sign
    pow         0      1
s       - Different sign
    pow         0      1
```

Listing 6.134: *`error_report` evaluation function excerpt for the `cuda` FPU with the `tPdn` testset.*

6.9 Optimisation framework

In this section, we demonstrate how the optimisation framework can be used to study the influence of compiler options on IEEE-conformity and application performance. We start with a short description of the user environments in which the evaluation framework was executed before discussing a two-step process of quickly selecting promising compiler options and applying these to a full test run. We conclude this section with two larger examples that make use of the two external applications which `IeeeCC754++` supplies in order to retrieve performance results.

6.9.1 User environments

The following two user environments were used in this section (both of which have been used before in this chapter, cf. Sections 6.1.1 and 6.3.1):

- An x86 workstation featuring an Intel Core i7-4770 processor (Haswell microarchitecture, cf. [INT13a]) with openSUSE 13.1 (`x86_64` with Linux kernel `3.12.62-55`). This platform is used as a default environment, i. e. it is paired with the `main` FPU of the `default` architecture.
- An x86 server with an Intel Xeon E5-2620 v4 CPU (Broadwell microarchitecture, cf. [INT16b]) that is running CentOS Linux 7.3.1611 (`x86_64` with Linux kernel `3.10.0-514.26.1.el7.x86_64`). On this platform, the `x86` architecture is used with the `main` FPU.

All test runs were performed with the `t3d` testset.

6.9.2 Two-step process

When a large number of compiler options needs to be evaluated with regard to floating-point conformity and application performance, it is usually not feasible to feed all these options into the optimisation framework and execute a test run that performs IEEE-conformity testing with `IeeeCC754++` and performance retrieval of the target application due to the latter unnecessarily slowing down the evaluation process. Rather, it is advisable to only execute `IeeeCC754++` with the different combinations of compiler options and omit the execution of the target application in this first step. The generated IEEE-conformity results can then be used to select a promising subset of compiler options and feed these into another optimisation framework run, this time enabling the external application and retrieving the runtimes resulting from the different compiler switch combinations.

For the following example demonstrating the two-step process, we use the default environment listed above together with `gcc` 7.2 as compiler. For the first step, the opt task file shown in Listing 6.135 was used.

```
ARCH = default
FPU = main
COMPILERS = gcc-7.2
MODULE = compiler/[cn]/[cv]
TESTSET = t3d
LEVEL1 = -mfpmath=387 -mfpmath=sse -mavx
LEVEL2 = -O1 -O3
LEVEL3 = -fno-rounding-math -funsafe-math-optimizations
COMBINATIONSLEVEL = 1
USE_EXTERNAL_APP = no
FITNESS = weighted
```

Listing 6.135: *Opt task file* `default_step_one.opt`.

On level 1, the three different FPUs that are contained in every current x86 CPU are targeted: x87, SSE, and AVX. On the second level, the common optimisation compiler switches `-O1` and `-O3` are used. Finally, the last (third) level shows the two options `-fno-rounding-math` and `-funsafe-math-optimizations` enabling the compiler to perform optimisations able to change floating-point semantics with regard to rounding behaviour and all aspects of floating-point behaviour, respectively. Listing 6.136 shows the rather lengthy output of the corresponding test run, evaluated with the `weighted` fitness function (see Section 3.5.2).

```
Format: name  succ_rate err_rate noexp_rate  runtime  overall succ err noexp | fitness [arch fpu
     compiler] options

   weighted

default_gcc72_main_set00001   99.86   0.14 100.00    0.00  16661  16638     23  16661 |  53.93
     [default main gcc-7.2]
default_gcc72_main_set00019   99.86   0.14 100.00    0.00  16661  16638     23  16661 |  53.93
     [default main gcc-7.2] -mfpmath=sse
default_gcc72_main_set00028   99.86   0.14 100.00    0.00  16661  16638     23  16661 |  53.93
     [default main gcc-7.2] -mavx
default_gcc72_main_set00029   98.58   1.42 100.00    0.00  16661  16424    237  16661 |  53.29
     [default main gcc-7.2] -mavx -O1
default_gcc72_main_set00030   98.58   1.42 100.00    0.00  16661  16424    237  16661 |  53.29
     [default main gcc-7.2] -mavx -O1 -fno-rounding-math
default_gcc72_main_set00033   98.58   1.42 100.00    0.00  16661  16424    237  16661 |  53.29
     [default main gcc-7.2] -mavx -O3
default_gcc72_main_set00034   98.58   1.42 100.00    0.00  16661  16424    237  16661 |  53.29
     [default main gcc-7.2] -mavx -O3 -fno-rounding-math
default_gcc72_main_set00010   98.37   1.63  99.65    0.00  16661  16389    272  16603 |  53.18
     [default main gcc-7.2] -mfpmath=387
default_gcc72_main_set00002   98.21   1.79  99.63    0.00  16661  16363    298  16600 |  53.11
     [default main gcc-7.2] -O1
default_gcc72_main_set00003   98.21   1.79  99.63    0.00  16661  16363    298  16600 |  53.11
     [default main gcc-7.2] -O1 -fno-rounding-math
default_gcc72_main_set00006   98.21   1.79  99.63    0.00  16661  16363    298  16600 |  53.11
     [default main gcc-7.2] -O3
default_gcc72_main_set00007   98.21   1.79  99.63    0.00  16661  16363    298  16600 |  53.11
     [default main gcc-7.2] -O3 -fno-rounding-math
default_gcc72_main_set00020   98.21   1.79  99.63    0.00  16661  16363    298  16600 |  53.11
     [default main gcc-7.2] -mfpmath=sse -O1
default_gcc72_main_set00021   98.21   1.79  99.63    0.00  16661  16363    298  16600 |  53.11
     [default main gcc-7.2] -mfpmath=sse -O1 -fno-rounding-math
default_gcc72_main_set00024   98.21   1.79  99.63    0.00  16661  16363    298  16600 |  53.11
     [default main gcc-7.2] -mfpmath=sse -O3
default_gcc72_main_set00025   98.21   1.79  99.63    0.00  16661  16363    298  16600 |  53.11
     [default main gcc-7.2] -mfpmath=sse -O3 -fno-rounding-math
default_gcc72_main_set00011   96.27   3.73  98.81    0.00  16661  16039    622  16463 |  52.13
     [default main gcc-7.2] -mfpmath=387 -O1
default_gcc72_main_set00012   96.27   3.73  98.81    0.00  16661  16039    622  16463 |  52.13
     [default main gcc-7.2] -mfpmath=387 -O1 -fno-rounding-math
default_gcc72_main_set00015   96.27   3.73  98.81    0.00  16661  16039    622  16463 |  52.13
     [default main gcc-7.2] -mfpmath=387 -O3
default_gcc72_main_set00016   96.27   3.73  98.81    0.00  16661  16039    622  16463 |  52.13
     [default main gcc-7.2] -mfpmath=387 -O3 -fno-rounding-math
default_gcc72_main_set00013   94.72   5.28  97.31    0.00  16661  15781    880  16213 |  51.36
     [default main gcc-7.2] -mfpmath=387 -O1 -funsafe-math-optimizations
default_gcc72_main_set00017   94.72   5.28  97.31    0.00  16661  15781    880  16213 |  51.36
     [default main gcc-7.2] -mfpmath=387 -O3 -funsafe-math-optimizations
default_gcc72_main_set00014   94.72   5.28  97.31    0.00  16661  15781    880  16213 |  47.36
     [default main gcc-7.2] -mfpmath=387 -O1 -fno-rounding-math -funsafe-math-optimizations
default_gcc72_main_set00018   94.72   5.28  97.31    0.00  16661  15781    880  16213 |  47.36
     [default main gcc-7.2] -mfpmath=387 -O3 -fno-rounding-math -funsafe-math-optimizations
default_gcc72_main_set00031   72.55  27.45  77.27    0.00  16661  12087   4574  12874 |  40.27
     [default main gcc-7.2] -mavx -O1 -funsafe-math-optimizations
default_gcc72_main_set00035   72.55  27.45  77.27    0.00  16661  12087   4574  12874 |  40.27
     [default main gcc-7.2] -mavx -O3 -funsafe-math-optimizations
default_gcc72_main_set00004   72.22  27.78  76.93    0.00  16661  12032   4629  12817 |  40.11
     [default main gcc-7.2] -O1 -funsafe-math-optimizations
default_gcc72_main_set00005   72.22  27.78  76.93    0.00  16661  12032   4629  12817 |  40.11
     [default main gcc-7.2] -O1 -fno-rounding-math -funsafe-math-optimizations
default_gcc72_main_set00008   72.22  27.78  76.93    0.00  16661  12032   4629  12817 |  40.11
     [default main gcc-7.2] -O3 -funsafe-math-optimizations
default_gcc72_main_set00009   72.22  27.78  76.93    0.00  16661  12032   4629  12817 |  40.11
     [default main gcc-7.2] -O3 -fno-rounding-math -funsafe-math-optimizations
```

```
default_gcc72_main_set00022   72.22  27.78  76.93    0.00  16661  12032   4629  12817 |  40.11
     [default main gcc-7.2] -mfpmath=sse -O1 -funsafe-math-optimizations
default_gcc72_main_set00026   72.22  27.78  76.93    0.00  16661  12032   4629  12817 |  40.11
     [default main gcc-7.2] -mfpmath=sse -O3 -funsafe-math-optimizations
default_gcc72_main_set00032   72.55  27.45  77.27    0.00  16661  12087   4574  12874 |  36.27
     [default main gcc-7.2] -mavx -O1 -fno-rounding-math -funsafe-math-optimizations
default_gcc72_main_set00036   72.55  27.45  77.27    0.00  16661  12087   4574  12874 |  36.27
     [default main gcc-7.2] -mavx -O3 -fno-rounding-math -funsafe-math-optimizations
default_gcc72_main_set00023   72.22  27.78  76.93    0.00  16661  12032   4629  12817 |  36.11
     [default main gcc-7.2] -mfpmath=sse -O1 -fno-rounding-math -funsafe-math-optimizations
default_gcc72_main_set00027   72.22  27.78  76.93    0.00  16661  12032   4629  12817 |  36.11
     [default main gcc-7.2] -mfpmath=sse -O3 -fno-rounding-math -funsafe-math-optimizations
```

Listing 6.136: *Output of* `opt.py default_step_one.opt`.

Without optimisations, SSE and AVX yield identical results, directly followed by the slightly worse results for the x87 FPU. When paired with additional compiler options, the results for x87 rapidly deteriorate, whereas SSE and AVX stay on the same IEEE-conformity level. For the step two run, we choose the AVX FPU since it is the newer and potentially better performing FPU. This option is needed for every executable built during the optimisation framework run, and we therefore add it to the opt task variable `CXXFLAGS`.

We omit both of the additional options tested in step one on level 3 since the first has no visible effect on floating-point results (and does not promise performance gains) and the second drastically reduces the level of IEEE-conformity. Instead of these options, we add the switch `-funroll-loops` which does not affect the semantics of the test vector execution during an `IeeeCC754++` run, but can potentially result in a tremendous performance boost for `sixloops` which is used as external application. Listing 6.137 shows the resulting opt task file for step two.

```
ARCH = default
FPU = main
COMPILERS = gcc-7.2
MODULE = compiler/[cn]/[cv]
TESTSET = t3d
LEVEL1 = -O1 -O3
LEVEL2 = -funroll-loops
LEVEL3 =
COMBINATIONSLEVEL = 1
CXXFLAGS = -mavx
APP_BUILD   = sixloops_mod_build.sh
APP_EXEC    = sixloops_mod_execute.sh
APP_REPEATS = 3
APP_TIMING  = external
USE_EXTERNAL_APP = yes
FITNESS = weighted
```

Listing 6.137: *Opt task file* `default_step_two.opt`.

Due to the reduced set of compiler options, only six sets of compiler options need to be executed. The results for the `weighted` fitness function are shown in Listing 6.138.

```
default_gcc72_main_set00005   98.58   1.42 100.00    8.60  16661  16424    237  16661 |  79.89
     [default main gcc-7.2] -O3
default_gcc72_main_set00006   98.58   1.42 100.00    8.31  16661  16424    237  16661 |  77.93
     [default main gcc-7.2] -O3 -funroll-loops
default_gcc72_main_set00003   98.58   1.42 100.00   27.54  16661  16424    237  16661 |  76.80
     [default main gcc-7.2] -O1
default_gcc72_main_set00004   98.58   1.42 100.00   24.53  16661  16424    237  16661 |  75.29
     [default main gcc-7.2] -O1 -funroll-loops
default_gcc72_main_set00001   99.86   0.14 100.00  184.04  16661  16638     23  16661 |  51.95
     [default main gcc-7.2]
default_gcc72_main_set00002   99.86   0.14 100.00  184.15  16661  16638     23  16661 |  51.93
     [default main gcc-7.2] -funroll-loops
```

Listing 6.138: *Output of* `opt.py default_step_two.opt`.

The runtimes for the execution of the `sixloops` application (which are shown in the fifth column) demonstrate the huge difference between optimised and non-optimised executables. Furthermore, the option `-funroll-loops` further decreases the runtimes, if only slightly. The reason for the `weighted` fitness function favouring the versions without this parameter lie in the selection of the weight factors which give a rather high priority to the "lower number of options" weight factor (cf. Section 3.5.2). However, since adding `-funroll-loops` does not change the IEEE-conformity results, the user can still choose the option combination `-O3 -funroll-loops` to compile the target application `sixloops`.

6.9.3 Example run with `sixloops`

Listing 6.139 shows the opt task file used for testing the influence of some optimisation options on the `sixloops` example application (cf. Section 3.5.3) used in the server user environment described in Section 6.9.1. For this optimisation framework run, we used `gcc` 4.8 as compiler. Note that the choice of FPU that is used to execute floating-point operations is left to the compiler.

```
ARCH = x86
FPU = main
COMPILERS = gcc-4.8
MODULE = compiler/[cn]/[cv]
TESTSET = t3d
LEVEL1 = -O1 -O3
LEVEL2 = -funroll-loops -funsafe-math-optimizations -ffast-math -frounding-math
LEVEL3 =
COMBINATIONSLEVEL = 1
APP_BUILD   = sixloops_mod_build.sh
APP_EXEC    = sixloops_mod_execute.sh
APP_REPEATS = 3
APP_TIMING  = external
USE_EXTERNAL_APP = yes
FITNESS = weighted
```

Listing 6.139: *Opt task file* `sixloops.opt`.

The output of the `weighted` fitness function for the optimisation framework run with this opt task file is shown in Listing 6.140.

```
x86_gcc48_main_set00034   93.34   6.66  98.43   18.86  16581  15476   1105  16320 |  79.20 [x86 main
     gcc-4.8] -O3 -funroll-loops
x86_gcc48_main_set00042   93.34   6.66  98.43   18.86  16581  15476   1105  16320 |  79.20 [x86 main
     gcc-4.8] -O3 -funroll-loops -frounding-math
x86_gcc48_main_set00041   93.34   6.66  98.43   20.20  16581  15476   1105  16320 |  79.02 [x86 main
     gcc-4.8] -O3 -frounding-math
x86_gcc48_main_set00033   93.34   6.66  98.43   20.31  16581  15476   1105  16320 |  79.01 [x86 main
     gcc-4.8] -O3
x86_gcc48_main_set00018   93.34   6.66  98.43   31.23  16581  15476   1105  16320 |  77.58 [x86 main
     gcc-4.8] -O1 -funroll-loops
x86_gcc48_main_set00026   93.34   6.66  98.43   31.72  16581  15476   1105  16320 |  77.51 [x86 main
     gcc-4.8] -O1 -funroll-loops -frounding-math
x86_gcc48_main_set00025   93.34   6.66  98.43   34.17  16581  15476   1105  16320 |  77.19 [x86 main
     gcc-4.8] -O1 -frounding-math
x86_gcc48_main_set00017   93.34   6.66  98.43   34.51  16581  15476   1105  16320 |  77.15 [x86 main
     gcc-4.8] -O1
x86_gcc48_main_set00040   73.58  26.42  78.31   18.75  16581  12200   4381  12985 |  69.33 [x86 main
     gcc-4.8] -O3 -funroll-loops -funsafe-math-optimizations -ffast-math
x86_gcc48_main_set00046   73.58  26.42  78.31   18.76  16581  12200   4381  12985 |  69.33 [x86 main
     gcc-4.8] -O3 -funroll-loops -ffast-math -frounding-math
x86_gcc48_main_set00038   73.58  26.42  78.31   18.87  16581  12200   4381  12985 |  69.32 [x86 main
     gcc-4.8] -O3 -funroll-loops -ffast-math
x86_gcc48_main_set00047   73.58  26.42  78.31   20.23  16581  12200   4381  12985 |  69.14 [x86 main
     gcc-4.8] -O3 -funsafe-math-optimizations -ffast-math -frounding-math
x86_gcc48_main_set00037   73.58  26.42  78.31   20.26  16581  12200   4381  12985 |  69.14 [x86 main
     gcc-4.8] -O3 -ffast-math
x86_gcc48_main_set00039   73.58  26.42  78.31   20.33  16581  12200   4381  12985 |  69.13 [x86 main
     gcc-4.8] -O3 -funsafe-math-optimizations -ffast-math
x86_gcc48_main_set00045   73.58  26.42  78.31   21.59  16581  12200   4381  12985 |  68.96 [x86 main
     gcc-4.8] -O3 -ffast-math -frounding-math
x86_gcc48_main_set00024   73.58  26.42  78.31   31.15  16581  12200   4381  12985 |  67.71 [x86 main
     gcc-4.8] -O1 -funroll-loops -funsafe-math-optimizations -ffast-math
x86_gcc48_main_set00022   73.58  26.42  78.31   31.19  16581  12200   4381  12985 |  67.71 [x86 main
     gcc-4.8] -O1 -funroll-loops -ffast-math
x86_gcc48_main_set00030   73.58  26.42  78.31   31.44  16581  12200   4381  12985 |  67.67 [x86 main
     gcc-4.8] -O1 -funroll-loops -ffast-math -frounding-math
x86_gcc48_main_set00023   73.58  26.42  78.31   31.53  16581  12200   4381  12985 |  67.66 [x86 main
     gcc-4.8] -O1 -funsafe-math-optimizations -ffast-math
x86_gcc48_main_set00031   73.58  26.42  78.31   31.54  16581  12200   4381  12985 |  67.66 [x86 main
     gcc-4.8] -O1 -funsafe-math-optimizations -ffast-math -frounding-math
x86_gcc48_main_set00021   73.58  26.42  78.31   31.81  16581  12200   4381  12985 |  67.62 [x86 main
     gcc-4.8] -O1 -ffast-math
x86_gcc48_main_set00029   73.58  26.42  78.31   34.28  16581  12200   4381  12985 |  67.30 [x86 main
     gcc-4.8] -O1 -ffast-math -frounding-math
x86_gcc48_main_set00036   68.03  31.97  75.62   18.80  16581  11280   5301  12539 |  66.55 [x86 main
     gcc-4.8] -O3 -funroll-loops -funsafe-math-optimizations
x86_gcc48_main_set00044   68.03  31.97  75.62   18.87  16581  11280   5301  12539 |  66.54 [x86 main
     gcc-4.8] -O3 -funroll-loops -funsafe-math-optimizations -frounding-math
x86_gcc48_main_set00035   68.03  31.97  75.62   20.64  16581  11280   5301  12539 |  66.31 [x86 main
     gcc-4.8] -O3 -funsafe-math-optimizations
x86_gcc48_main_set00043   68.03  31.97  75.62   20.90  16581  11280   5301  12539 |  66.28 [x86 main
     gcc-4.8] -O3 -funsafe-math-optimizations -frounding-math
x86_gcc48_main_set00027   68.03  31.97  75.62   31.28  16581  11280   5301  12539 |  64.92 [x86 main
     gcc-4.8] -O1 -funsafe-math-optimizations -frounding-math
x86_gcc48_main_set00019   68.03  31.97  75.62   31.63  16581  11280   5301  12539 |  64.87 [x86 main
     gcc-4.8] -O1 -funsafe-math-optimizations
x86_gcc48_main_set00028   68.03  31.97  75.62   32.52  16581  11280   5301  12539 |  64.76 [x86 main
     gcc-4.8] -O1 -funroll-loops -funsafe-math-optimizations -frounding-math
x86_gcc48_main_set00020   68.03  31.97  75.62   33.95  16581  11280   5301  12539 |  64.57 [x86 main
     gcc-4.8] -O1 -funroll-loops -funsafe-math-optimizations
x86_gcc48_main_set00048   73.58  26.42  78.31   18.71  16581  12200   4381  12985 |  64.34 [x86 main
     gcc-4.8] -O3 -funroll-loops -funsafe-math-optimizations -ffast-math -frounding-math
x86_gcc48_main_set00032   73.58  26.42  78.31   32.50  16581  12200   4381  12985 |  62.53 [x86 main
     gcc-4.8] -O1 -funroll-loops -funsafe-math-optimizations -ffast-math -frounding-math
x86_gcc48_main_set00002   94.63   5.37  98.43  223.78  16581  15690    891  16320 |  53.01 [x86 main
     gcc-4.8] -funroll-loops
x86_gcc48_main_set00001   94.63   5.37  98.43  226.41  16581  15690    891  16320 |  52.67 [x86 main
     gcc-4.8]
x86_gcc48_main_set00010   94.63   5.37  98.43  226.64  16581  15690    891  16320 |  52.64 [x86 main
     gcc-4.8] -funroll-loops -frounding-math
x86_gcc48_main_set00009   94.63   5.37  98.43  227.49  16581  15690    891  16320 |  52.53 [x86 main
     gcc-4.8] -frounding-math
x86_gcc48_main_set00012   69.13  30.87  75.97  223.12  16581  11463   5118  12596 |  40.35 [x86 main
     gcc-4.8] -funroll-loops -funsafe-math-optimizations -frounding-math
x86_gcc48_main_set00015   68.43  31.57  75.97  223.25  16581  11347   5234  12596 |  39.99 [x86 main
     gcc-4.8] -funsafe-math-optimizations -ffast-math -frounding-math
x86_gcc48_main_set00006   68.43  31.57  75.97  223.71  16581  11347   5234  12596 |  39.93 [x86 main
     gcc-4.8] -funroll-loops -ffast-math
x86_gcc48_main_set00014   68.43  31.57  75.97  224.18  16581  11347   5234  12596 |  39.87 [x86 main
     gcc-4.8] -funroll-loops -ffast-math -frounding-math
```

```
x86_gcc48_main_set00008   68.43  31.57  75.97  224.78  16581  11347   5234  12596 |  39.79 [x86 main
     gcc-4.8] -funroll-loops -funsafe-math-optimizations -ffast-math
x86_gcc48_main_set00005   68.43  31.57  75.97  225.05  16581  11347   5234  12596 |  39.75 [x86 main
     gcc-4.8] -ffast-math
x86_gcc48_main_set00003   69.13  30.87  75.97  227.88  16581  11463   5118  12596 |  39.73 [x86 main
     gcc-4.8] -funsafe-math-optimizations
x86_gcc48_main_set00011   69.13  30.87  75.97  228.39  16581  11463   5118  12596 |  39.66 [x86 main
     gcc-4.8] -funsafe-math-optimizations -frounding-math
x86_gcc48_main_set00004   69.13  30.87  75.97  229.13  16581  11463   5118  12596 |  39.57 [x86 main
     gcc-4.8] -funroll-loops -funsafe-math-optimizations
x86_gcc48_main_set00007   68.43  31.57  75.97  226.63  16581  11347   5234  12596 |  39.54 [x86 main
     gcc-4.8] -funsafe-math-optimizations -ffast-math
x86_gcc48_main_set00016   68.43  31.57  75.97  226.75  16581  11347   5234  12596 |  39.53 [x86 main
     gcc-4.8] -funroll-loops -funsafe-math-optimizations -ffast-math -frounding-math
x86_gcc48_main_set00013   68.43  31.57  75.97  226.84  16581  11347   5234  12596 |  39.52 [x86 main
     gcc-4.8] -ffast-math -frounding-math
```

Listing 6.140: *Output of* `opt.py sixloops.opt`.

Since using `-O3` does not seem to influence the IEEE-conformity in this environment, some of the corresponding test runs are found to be the best performing and at the same time yielding the second best success rates. Of the highest rated compiler option combinations, the one with `-funroll-loops` as second option besides `-O3` is rated best (tying with the three option combination `-O3 -funroll-loops -funsafe-math-optimizations` which is placed after the former due to lexicographical sorting) and therefore suggested by the optimisation framework as best possible set of compiler options (for this application out of the supplied switches).

6.9.4 Example run with HPCG

We conclude the optimisation framework section with some results of testing with HPCG as external application (cf. Section 3.5.3). Listing 6.141 contains the settings used in this optimisation framework run which was again executed in the server environment. For this example, we only used the optimisation options `-O1` and `-O3` together with `clang` 5.0 and three versions of `gcc`. All floating-point operations are executed in the AVX SIMD unit.

```
ARCH = x86
FPU = main
TESTSET = t3d
COMPILERS = gcc-4.7 gcc-5.5 gcc-7.2 clang-5.0
MODULE = compiler/[cn]/[cv]
LEVEL1 = -O1 -O3
LEVEL2 =
LEVEL3 =
COMBINATIONSLEVEL =
CXXFLAGS = -mavx
APP_BUILD   = hpcg_mod_build_parallel.sh
APP_EXEC    = hpcg_mod_execute_mpi.sh
APP_REPEATS = 3
APP_TIMING  = external
USE_EXTERNAL_APP = yes
FITNESS = weighted
```

Listing 6.141: *Opt task file* `hpcg.opt`.

Listing 6.142 once again shows the output of the `weighted` fitness function for the corresponding optimisation framework run.

```
x86_gcc47_main_set00002     98.57   1.43 100.00   10.52  16581  16344    237  16581 |  63.74 [x86
     main gcc-4.7]   -O1
x86_gcc47_main_set00003     98.57   1.43 100.00   10.66  16581  16344    237  16581 |  63.52 [x86
     main gcc-4.7]   -O3
x86_clang50_main_set00011   94.54   5.46  98.42   10.50  16581  15676    905  16319 |  61.74 [x86
     main clang-5.0] -O1
x86_clang50_main_set00012   94.54   5.46  98.42   10.54  16581  15676    905  16319 |  61.69 [x86
     main clang-5.0] -O3
x86_gcc55_main_set00005     93.34   6.66  98.43   10.50  16581  15476   1105  16320 |  61.15 [x86
     main gcc-5.5]   -O1
x86_gcc55_main_set00006     93.34   6.66  98.43   10.57  16581  15476   1105  16320 |  61.05 [x86
     main gcc-5.5]   -O3
x86_gcc72_main_set00008     93.34   6.66  98.43   10.62  16581  15476   1105  16320 |  60.96 [x86
     main gcc-7.2]   -O1
x86_gcc72_main_set00009     93.34   6.66  98.43   10.66  16581  15476   1105  16320 |  60.91 [x86
     main gcc-7.2]   -O3
x86_clang50_main_set00010   94.54   5.46  98.42   19.91  16581  15676    905  16319 |  48.85 [x86
     main clang-5.0]
x86_gcc55_main_set00004     94.63   5.37  98.43   20.11  16581  15690    891  16320 |  48.59 [x86
     main gcc-5.5]
x86_gcc47_main_set00001     94.63   5.37  98.43   20.26  16581  15690    891  16320 |  48.37 [x86
     main gcc-4.7]
x86_gcc72_main_set00007     94.63   5.37  98.43   20.29  16581  15690    891  16320 |  48.31 [x86
     main gcc-7.2]
```

Listing 6.142: *Output of `opt.py hpcg.opt`.*

These results highlight the extent of the compiler's influence on floating-point behaviour and application performance when applying different optimisation levels. The executables without optimisation show about half the performance compared to the optimised executables, whereas the runtimes of the optimised executables are roughly identical regardless of optimisation level. However, the reason for the `weight` fitness function selecting the executables compiled with `gcc` 4.7 as the best combination of compiler and options lies in the (rather astonishing) fact that the IEEE-conformity level is higher than for all other combinations (even compared to the non-optimised version compiled with `gcc` 4.7).

In order to analyse the reasons, we compared the logfiles for sets 1 and 3. In the `-O3` case, exception flags for conversions involving integers and the round to integral operator are missing which increases the error rate compared to the non-optimised version. However, the latter exhibits a surprising number of errors in the remainder operator where an inexact exception is erroneously signalled and some values are incorrectly rounded. These errors disappear in the optimised executable, and since the amount of remainder errors far surpasses the number of conversion errors, the success rate for the optimised executable increases.

6.10 Result summary

In the previous sections, we took a rather detailed look at a selection of architecture ports provided by `IeeeCC754++`. We conclude this chapter with a summary of these results from a more high-level point of view.

6.10.1 Basic operations and conversions

Since the elementary operators which were discussed in Sections 4.3 to 4.6 and 6.8 are not required from IEEE 754-2008 conforming environments, we start the summary by only regarding results for the operators required by IEEE 754-2008, i. e. basic operations (including `fma`) and conversions.

Default environments

The testing of default environments showed that it can indeed be helpful not to rely on the assumption that any given floating-point environment is fully IEEE-conforming. Especially the default environment on ARMv6 exhibited a significant amount of errors (cf. Section 6.4.1). Also, the standard square root operator in the POWER8 default environment tested in Section 6.5.1 should be avoided. Overall, most default environments showed mainly errors in the conversions between decimal and binary numbers which, in addition, are usually compiler dependent.

One interesting aspect of our new `fma` test vectors was shown in Section 6.1.2: These test vectors can distinguish between an implementation which is actually fused (i. e. it does not round the intermediate result) and one which simply combines regular floating-point multiplications and additions.

Environments without errors

Our testing revealed that some of the FPUs tested in this chapter are fully IEEE-conforming in the sense that the execution of the test vectors yielded no errors, i. e. the returned floating-point were correctly rounded and the FPUs returned exactly the exceptions that were expected. These are the x86 `sse`, `avx`, and `avx512` FPUs (see Sections 6.3.1 and 6.3.2), the ARMv8-A `sve` FPU (keeping in mind that this FPU only supports the roundTowardZero mode for conversion to integers, cf. Section 6.4.2), and the SoftFloat software floating-point library (see Section 6.7.1). Additionally, the MPFR library also returned almost completely correct results with only small problems related to floating-point exceptions due to MPFR using a non-standard exception model with six exceptions. This results in slightly different behaviour for the invalid exception (cf. Section 6.7.2).

Environments with errors exclusively related to exceptions

In addition to the fully IEEE-conforming floating-point environments, there are some platforms that return correctly rounded floating-point numbers, but do not support exceptions (either not at all or not fully conforming). The two most notable environments in this category are the `cudai` FPU (cf. Section 6.6.1) and the ARMv8-A `asimd` FPU (cf. Section 6.4.2).

Environments fully supporting default rounding modes

Since most numerical algorithms do not make use of switching the rounding mode, but rely on roundTiesToEven being the default rounding mode, some platforms only provide support for roundTiesToEven (such as OpenCL or `Java`) or support all rounding modes, but yield correctly rounded results mostly for roundTiesToEven (i. e. roundTiesToEven is more carefully implemented on these platforms). Note that for conversion to integers, the results are usually truncated instead of rounded, thereby making roundTowardZero the default rounding mode for these operators.

The following environments can be placed in this category (IEEE-conforming results for the default rounding modes): the ARMv7-A `vfpv4` FPU (Section 6.4.1), the ARMv8-A `neon` and `neoni` FPUs (Section 6.4.2), the POWER8 `vsx` FPU (Section 6.5.1), and OpenCL (at least for the user environment tested in this thesis, cf. Section 6.6.2). Finally, the default environment of the tested ARMv6 platform (on a Raspberry Pi, cf. Section 6.4.1) almost belongs into this category: Test results showed that only a small number of errors occurred in roundTiesToEven mode (opposed to substantial error counts for the other rounding modes).

Environments with minor errors

We found a few environments yielding only a small (but non-zero) number of errors, such as the x86 `x87` FPU (see Section 6.3), the ARMv7-A `vfp` FPU (cf. Section 6.4.1), and the `Java` FPUs (Section 6.7.3). Therefore, these environments can be called “almost conforming”.

Environments not fully conforming to IEEE 754-2008

We conclude the summary for the basic operations and conversions with the platforms we found to be implemented in a not-so-conforming way. These include the ARMv7-A NEON FPUs (Section 6.4.1) and the Cell SPU (Section 6.5.2). For these environments, the deficiencies are the result of deliberate design decisions, such as not supporting subnormals or using the number format in an IEEE 754-2008 incompatible manner.

6.10.2 Elementary operators

Since the elementary operators are not mandatory for IEEE-conforming floating-point environments, only some of the tested environments include implementations for these operators. As another consequence, errors found in the elementary operators do not break conformity of the respective environment.

None of the tested FPUs were found to be returning fully conforming results. CRlibm and MPFR return correctly rounded floating-point numbers for all test vectors (and the operators implemented in these libraries) and show only

(minor) errors related to floating-point exception handling. In the other three FPUs supporting elementary operators (CUDA, `C99`, and `C++11`), some cases of incorrectly rounded floating-point numbers being returned were found, with the largest error count found for the CUDA implementation. However, all of these cases deviate from the correctly rounded result by only 1 ulp. We can therefore conclude that although the quality of the elementary operator implementation is not as high as for the basic arithmetic operators, it should be sufficient for most numerical applications which need to deal with floating-point inaccuracy on an algorithmic level anyway. Furthermore, MPFR and CRlibm prove that it is possible to properly implement correctly rounded elementary operators.

6.10.3 Some notes on applications

We conclude our summary with a few thoughts about the effect of a user environment's IEEE-conformity on two applications. In this section, we showed that on most platforms, IEEE 754-2008 support is usually quite good. However, some of the tested user environments can only be called compatible when ignoring floating-point exceptions, and some do not provide the four rounding modes required by IEEE 754-2008. The latter has severe consequences for using interval arithmetic in such an environment: Existing libraries cannot be used without modifications since switching rounding modes does not necessarily provide the correct rounding. [CFD08] demonstrates the effort needed to port the Boost interval library [BMP06] to NVidia GPUs using the CUDA toolkit.

In the last years, using mixed precision floating-point computations has gained significant momentum due to performance reasons (see e. g. [Hig15; But$^+$06]), with a latest focus on exploiting half precision support in floating-point hardware such as GPGPUs or some ARM processors especially in machine learning applications. Since higher precision (typically double) is used in critical parts of mixed precision algorithms, such as in computing the residuum of a linear solver which is particularly sensitive to rounding errors, one can afford the use of lower precision in other parts of the algorithms. Overall, this approach yields convergence behaviour identical to using computations which were performed solely in the highest precision, albeit with better performance. This model even allows for alleviation of the requirements for the IEEE-conformity of the smaller precision computations (such as for the PowerXCell 8i processor which uses FTZ and the roundTowardZero rounding mode for single precision computations in its SPUs, cf. Sections 5.4.2 and 6.5.2), since the exactness of the algorithm only relies on the exactness (and thus the IEEE-conformity) of the higher precision.

The second example shows the use of such a mixed precision algorithm on the PowerXCell 8i processor. [Nob11] discusses the implementation of an application from QCD (cf. e. g. [WIK17y]) on the QPACE supercomputer (see e. g. [Bai$^+$09; WIK17x]), which was successfully deployed by the SFB/TR55 [SFBTR]. In order to achieve good performance on the PowerXCell 8i processor and to avoid stagnation

in the iterative solver (due to floating-point accuracy problems), a robust algorithm making use of mixed precision had to be chosen, see also [FNZ12]. With this implementation, convergence of the solver and good application performance could be achieved despite the PowerXCell 8i processor not being fully conforming to IEEE 754-2008.

Summary and outlook

In this thesis, we presented `IeeeCC754++` and an accompanying set of tools that enable testing the conformity to IEEE 754-2008 of arbitrary platforms, with a special focus on the default user environment experienced by a researcher developing numerical applications. After briefly introducing the basics of floating-point numbers, the relevant standards, and considerations related to (floating-point) user environments, we discussed the established testing tool `IeeeCC754` which we chose as a base for our significantly extended new tool. With `IeeeCC754++`, we added support for IEEE 754-2008 in a number of ways, such as adding mandatory (`fma`) and recommended (elementary functions) operators, extending the collection of test vectors in the precision independent Coonen format, lifting precision restrictions that prevented testing of floating-point numbers in the half precision format and thereby enabling initial half precision support, and adapting the input and output facilities to reflect the new feature set. We introduced the evaluation and optimisation frameworks and the graphical application `IeeeCC754++LogViewer` that ease the analysis of different aspects of a possibly large number of floating-point environments. Furthermore, we implemented a substantial number of architecture ports and presented results for a selection of user environments, revealing overall that floating-point numbers are mostly supported in an IEEE-conforming way (at least when ignoring floating-point exceptions and, to a smaller degree, the switching of rounding modes).

In order to contribute to future IEEE-conformity testing, we finally presented facilities to extend our existing tool `IeeeCC754++` with additional arbitrary floating-point platforms by implementing custom architecture ports with moderate effort. We meticulously documented this process of adding such a custom architecture port to `IeeeCC754++` in Appendices A and B.

The scope of this thesis did not allow for testing a more comprehensive selection of floating-point environments, let alone complete coverage of currently available and widespread platforms. As a natural starting point for future work, we suggest applying our tool `IeeeCC754++` and the analysis techniques that we presented to the following (obviously incomplete list) of architectures and platforms in order to assess their IEEE-conformity:

- The x86-compatible EPYC line of server processors released by AMD (cf. [AMD17a]) and their desktop variant Ryzen (see [AMD17b]), especially in contrast to Intel server and desktop CPUs.
- GPGPUs by AMD such as FirePro [AMD17d] or Radeon [AMD17c] GPUs. Again, a direct comparison with results generated on GPGPUs by a competitor (in this case, NVidia) might yield interesting insight.
- Current processors or server architectures based on ARMv8-A ISA, such as Cavium ThunderX [Cav17; Gel16] or Qualcomm Centriq 2400 [Qua17; Mor17b] (or any processor based on ARMv8-A, such as Cortex-A57, Cortex-A73, Cortex-A75). In particular, our implementation of the ARM SVE unit should be applied to actual hardware implementing this extension as soon as it gets available.

Especially when considering that the development and deployment of new floating-point hardware and software, be it CPUs, FPUs, accelerators, or software libraries, is an ever ongoing process, it is clear that no tool can ever be complete in the sense that it provides support for all existing architectures. As a consequence, one of the most obvious ways to expand upon this work represents the addition of new architecture ports for platforms that are either already deployed in HPC or have the potential of becoming significant future deployments. In the following, we list a few such platforms:

- POWER9 [Sad+17], whose rollout has already started (see [Mor17c]). A dedicated `power9` architecture port might not be necessary since testing could be potentially performed with the `ppc` port.
- Processors based on MIPS ISA, cf. [MIP17].
- New HPC accelerators such as Pezy SC-2 PCI-X cards [Sch17] (which are based on MIPS ISA, see [Ber17]) or NEC SX-Aurora which features a multicore vector processor, cf. [NEC17; Mor17a].
- It would be very interesting to evaluate the IEEE-conformity of the current number #1 system on the Top500, the China-built Sunway TaihuLight (cf. [TOP500a; WIK17ag]) which is based on a 64 bit RISC architecture (see [WIK17af]). However, not too much is known about the architecture of the

underlying Sunway SW26010 processor (cf. [WIK17ah]), and it might be impossible to gain access to a computing system featuring this processor.

We conclude this thesis by pointing out a number of additional future research topics related to and based on our contributions:

- *Improved support for half precision*: As discussed in Section 3.1.11, the current testsets include test vectors which might not be applicable to the half precision format. For full support of this format, these test vectors need to be identified and excluded. Additionally, the grammar of the precision independent Coonen format and `IeeeCC754++`'s parser need to be extended in a way which makes it possible to exclude test vectors from being applied to a specific precision, since currently it is only possible to mark test vectors to be used with either all or exactly one of the available precisions.

  Furthermore, some of the platforms already supplied by architecture ports implemented in `IeeeCC754++` support operands in half precision. Therefore, conversion between half precision and `IeeeCC754++`'s internal floating-point representation, as well as the available half precision operators, could be implemented for these platforms such as GPUs based on the NVidia Volta microarchitecture [NVi17b] (or generally for the CUDA and OpenCL ports), the AARCH64 port for processors featuring the ARMv8.2-FP16 extension (cf. [ARM17]), or Intel Xeon Phi processors codenamed Knights Mill (see e. g. [Smi16]).

- *Extending the test vector sets*: The initial testsets for the elementary functions which we included in this thesis mainly consist of simple test cases, all special cases listed in IEEE 754-2008, and known hard or worst to round cases for double precision. In order to provide IEEE-conformity testing for a more comprehensive set of precisions, especially for half and single precision, further research is needed to identify bad or worst cases for the elementary functions and encode these as test vectors for use with `IeeeCC754++`.

- *Support for IEEE 754-2018*: As briefly discussed in Section 1.2.4, IEEE 754-2008 is currently being revised in order to produce IEEE 754-2018 (the revision being necessary to conform to the IEEE rule of standards being valid for 10 years). Although all fundamental major changes have been postponed to the potential standard IEEE 754-2028, there are a number of smaller changes, most of which do not influence the conformity of existing floating-point implementations. The following changes are worth reviewing in the context of `IeeeCC754++` in order to enable IEEE 754-2018 support (see also [Hou17a]): Clause 9.2 of the (proposed) new standard adds the operators `asinPi` and `acosPi` which were missing in IEEE 754-2008, and for some operators (especially for the power operators), the specification of special cases concerning e. g. infinity and NaN handling has been clarified and

extended. Adapting `IeeeCC754++` to these changes and adding appropriate test vectors should be feasible, although finding hard or worst to round cases for the new trigonometric operators might require substantial research efforts.

The new clause 9.5 of IEEE 754-2018 adds augmented operators for addition, subtraction, and multiplication which not only return the correctly rounded result, but also the rounding error. It is unclear whether it makes sense to integrate these operations into `IeeeCC754++`. Therefore, studying the applicability and potential use of integrating these operators into `IeeeCC754++` represents an interesting research topic.

Appendix A

The IeeeCC754++ build system

In order to support all of the new features introduced in Chapter 3, it was necessary to significantly restructure and extend the original `IeeeCC754` application.. As an established test program that underwent intensive development and testing over a long time span, `IeeeCC754` itself represents a stable and mature program. Therefore, the code was only changed where necessary to support new features or improve maintainability. Especially the internal data format to represent floating-point numbers, the syntax describing test vectors, the parser used to read (and write) those test vectors, and the test execution and analysis engine were mostly left untouched, but carefully extended where necessary. For an overview of the new features, we refer to Section 3.1.

A.1 Changes to the code base

We start this chapter by giving a short overview of changes introduced into the `IeeeCC754` code base in order to provide a flexible and easily extensible code base for `IeeeCC754++`. Afterwards, the new build system is explained in detail, together with a description how to configure and build `IeeeCC754++` for a given floating-point environment. The process of extending `IeeeCC754++` with new architectures and FPUs (i. e. adding a new port) can be found in Appendix B.

In Chapter 3, we give a detailed description of `IeeeCC754++` and the newly introduced features. In particular, Section 3.1.5 explains the need to restructure `IeeeCC754` with the concepts *architecture* and *FPU* (cf. also Definitions 3.2 and 3.3). This approach of separating the environment dependent parts into architectures and FPUs enables independent development of the actual testing core including the handling of the different modes and input/output formats on the one hand and extensions with new specific FPU ports on the other hand. In order to explain the implementation of `IeeeCC754++`'s base functionality, we start by describing the origins of `IeeeCC754`.

A.1.1 IeeeCC754 code structure

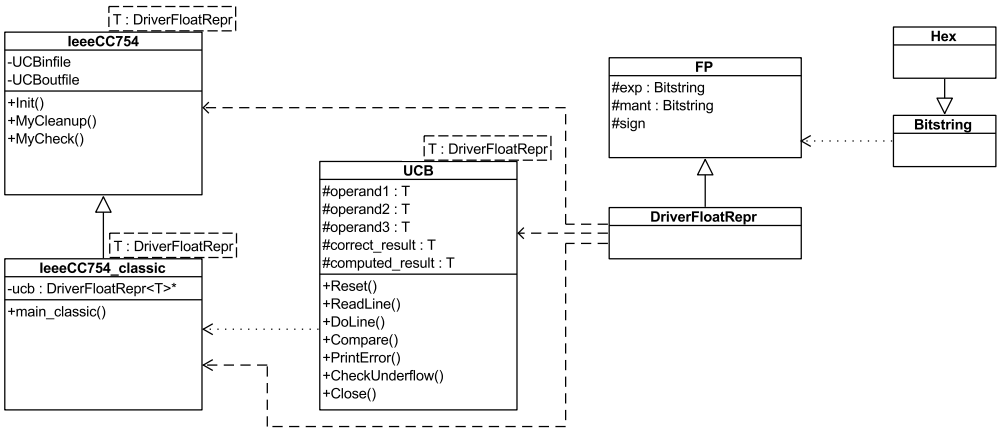

Figure A.1: *Class hierarchy of* `IeeeCC754`

Figure A.1 shows the basic code structure of `IeeeCC754`, albeit *after* restructuring the code in preparation for the extensions described in Chapter 3. It consists of basically two sets of classes: one for the testing core including file handling, test vector parsing, and evaluation of the testing results, and one implementing an internal data format for floating-point numbers.

Almost all code dealing with the classic modes (cf. Section 3.3.1, especially test vector parsing and file handling, is implemented in `IeeeCC754_classic<T>` which is derived from `IeeeCC754<T>`. The latter class consists of some variables and functions which are common to all testing modes, including the (new) extended ones. Inside `IeeeCC754_classic<T>`, the actual execution of test vectors, i. e. translating test vectors from UCB to environment format, executing an operation, and evaluating the results, is performed by an instance of `UCB<T>`. To achieve this, it contains variables for the (up to three) floating-point operands, the correct result as well as the returned result, and information about expected and returned exception flags.

The classes implementing the testing core take a template parameter `T` that describes a user environment specific implementation of `IeeeCC754`'s internal data format which is called `DriverFloatRepr`. It is derived from the class `FP` which contains the implementation of `IeeeCC754`'s internal representation of a floating-point number, including a set of access functions and methods to check if the current floating-point number is a normalised, subnormal or infinite number or if it is an NaN. `DriverFloatRepr` adds implementations of the actual operators to be tested, code for the description of the current FPU (e. g. whether it is a vectorised unit), and functions that interact with the underlying floating-point environment.

Inside of `FP`, the values of exponent and significand are stored in two bit strings which are implemented in the class `Bitstring`. The derived class `Hex` adds a

hexadecimal representation of a `Bitstring` that is needed inside the test vector parser. In addition, `FP` contains variables for the sign, the hidden bit (i. e. whether one is used for the current format), and sizes of exponent and significand.

A.1.2 `IeeeCC754++` code structure

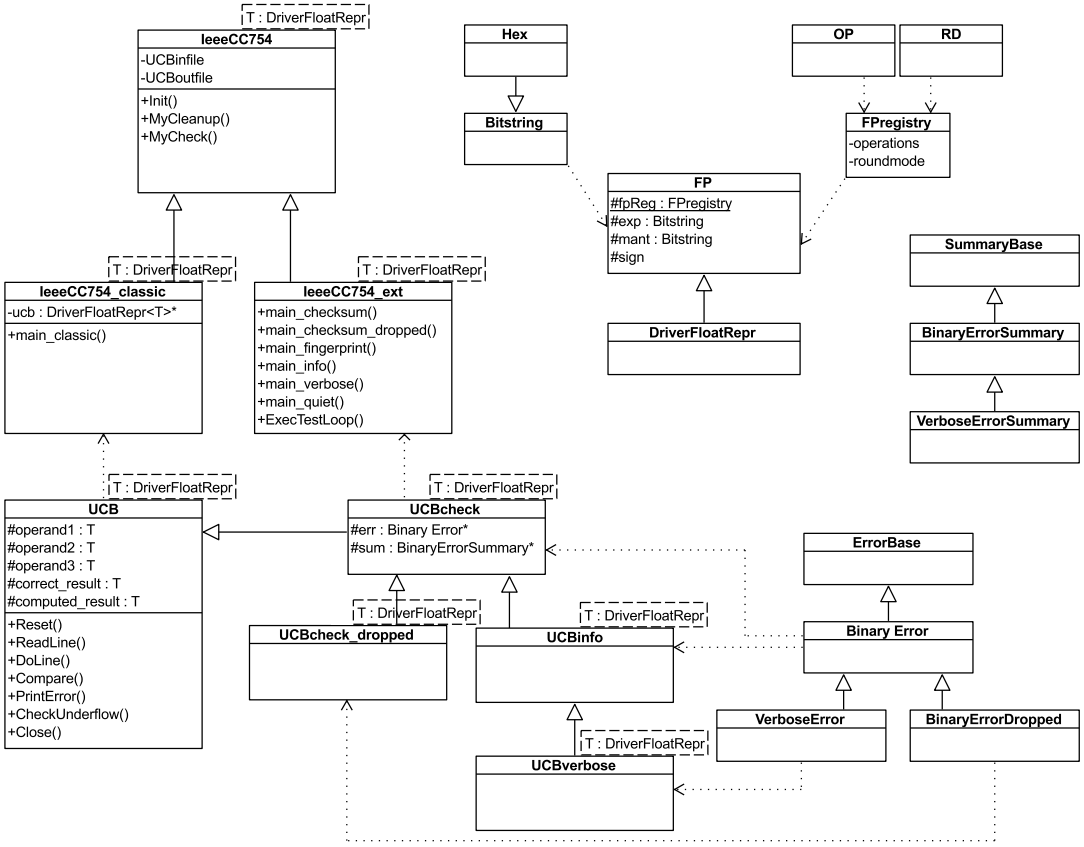

Figure A.2: *Class hierarchy of `IeeeCC754++`*

The class hierarchy shown in Figure A.1 can easily be extended with new functionality like the testing modes described in Section 3.3 by deriving from the corresponding classes. The extended class hierarchy is shown in Figure A.2. As in `IeeeCC754`, the core class handling command line parsing, file handling, and calling the respective modes is derived from `IeeeCC754<T>`. The new class `IeeeCC754_ext<T>` contains functions that implement all extended modes. Each of these sets up an appropriate instance of a class derived from `UCB<T>` and calls `ExecTestLoop()` to execute the tests.

The actual testing process is mostly identical in the classic and extended modes (which are implemented in `IeeeCC754_classic<T>` and `IeeeCC754_ext<T>`), the main differences lying in the output formats. These are supported by introducing two base classes `ErrorBase` and `SummaryBase` which describe errors and the overall

summary. UCBcheck<T> (which is derived from UCB<T>) and all classes derived from it include an instance of an error and a summary class derived from these. This means each of the extended modes is represented as a combination of a main_ function in IeeeCC754_ext<T> together with a testing class, an error class, and a summary class, derived from UCB<T>, ErrorBase, and SummaryBase, respectively. Figure A.2 depicts the relation between testing and error classes with dotted lines. Note that the relation of the summary classes with the testing classes has been omitted for increased clarity. The realisation of the different extended testing modes is shown in Table A.1.

Mode	Method	testing class	Error class	Summary class
-s	main_checksum	UCBcheck	BinaryError	BinaryErrorSummary
-d	main_checksum_dropped	UCBcheck_dropped	BinaryErrorDropped	BinaryErrorSummary
-h	main_fingerprint	UCBcheck	BinaryError	BinaryErrorSummary
-i	main_info	UCBinfo	BinaryError	BinaryErrorSummary
-v	main_verbose	UCBverbose	VerboseError	VerboseErrorSummary
-q	main_quiet	UCBcheck	BinaryError	BinaryErrorSummary

Table A.1: *Relation between testing modes and the classes used to implement these.*

All classes derived from IeeeCC754<T> and UCB<T> need knowledge about the underlying floating-point format. This is again achieved by implementing an FPU of the current architecture as a class derived from DriverFloatRepr.

Since IeeeCC754++ extends the number of available operators considerably (see Section 3.1.11) and many FPUs support only a subset of these operations, a mechanism is needed to exclude operations from testing. Furthermore, not all FPUs support all rounding modes required in IEEE 754-2008, thus rendering it pointless to test these rounding modes. IeeeCC754++ enables the exclusion of operations or rounding modes via a static member of DriverFloatRepr called FPregistry. Before executing a test vector, it is checked whether the operation and rounding mode under investigation are registered inside this class, and only then the operation is actually performed. The description of known operations and rounding modes is contained in the classes OP and RD. A detailed discussion of the classes DriverFloatRepr and FPregistry is given in Appendix B together with the steps needed to implement a custom architecture or FPU.

A.2 The build system

As described in Section 3.1.6, the IeeeCC754++ has been implemented using Autotools. In order to use Autotools' flexible configuration facilities as efficiently as possible while at the same time supporting an arbitrary number of architectures and FPUs (and enabling extending IeeeCC754++ with further architectures and FPUs), the source code had to be restructured (see Section A.1) and the file layout needed to be adapted.

Figure A.3 shows a rough overview of the new file structure:

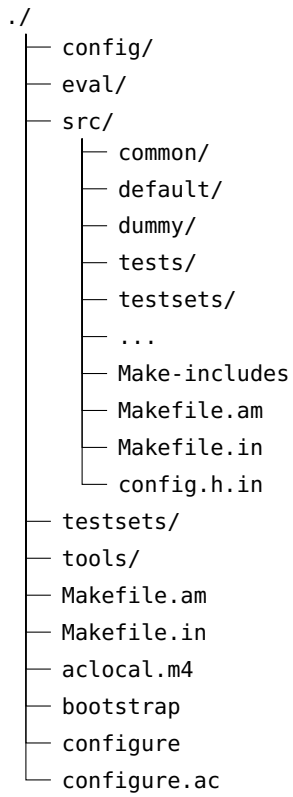

Figure A.3: *Basic code structure of* `IeeeCC754++`

- `configure.ac` and `Makefile.am` are the main Autoconf and Automake configuration files and contain the setup and definitions for the build system.
- The subdirectory `config/` contains further macros needed by Autotools.
- The script `bootstrap` takes these files (and further files defined in subdirectories such as `src/Makefile.am`) and generates the main `configure` script as well as all `Makefile.in` templates. Some additional files like `aclocal.m4` and `src/config.h.in` (the latter containing macros needed by the `IeeeCC754++` source files) are also generated in this step.

  When `configure` is called, it uses these `.in` files to create the `Makefile`s that are used by `make` to build `IeeeCC754++`.
- The directory `src/` contains the actual `IeeeCC754++` source code. The files implementing the core functionality, such as parsing test vector files, executing the tests, and evaluating the results, can be found in `src/common/`. Architectures are implemented in individual subdirectories; Figure A.3 shows the directories `default/` and `dummy/` as examples. Inside these subdirectories,

the source files for the respective architecture and all corresponding FPUs are located, together with further files needed to set up the build for this architecture and potentially for interfacing custom libraries.

Additionally, `src/testsets/` contains the test vector files in Coonen format, and `src/tests/` includes some simple test programs (see below).

- The evaluation framework is implemented inside the directory `eval/`, together with the optimisation framework. For details, see Sections 3.4 and 3.5.
- The directory `testsets/` contains testsets in UCB format and the script `genUBB.sh` to generate collections of testsets containing only specific operations, cf. Section 4.7.
- Finally, the directory `tools/` contains some utility programs that are helpful when using `IeeeCC754++`: the source code of the Autotools packages (`autoconf`, `automake`, and `m4`) used to create the current versions of the `Makefile.in` files via `bootstrap` as well as the `IeeeCC754++LogViewer` program that can be used to view result files generated by the evaluation framework (cf. Section 3.4.3).

When an architecture is chosen to be built, possibly with one or more additional FPUs, the build system generates appropriate `Makefile`s that take care of generating the common code and code for the current architecture. This approach ensures efficient compilation resulting in an executable containing no unnecessary code. Furthermore, the build system attempts to support the user in setting up the architecture build by attempting to adapt necessary settings automatically, especially settings for compiler options or additional libraries needed to build `IeeeCC754++` for the current architecture. As these settings are highly dependent on the current combination of compiler, architecture, and FPUs, automatic detection will not always result in a successful build. Therefore, the `IeeeCC754++` provides means to specify further (or even completely custom) compiler options, see below.

During the configuration process, the build system tries to detect relevant parameters of the user environment, e. g. the presence of certain system libraries. The results of this detection are then used, together with the user choices for compiler, architecture, and FPU, to generate the file `src/config.h` which includes suitable macro and variable definitions which are needed inside the `IeeeCC754++` source files to compile correctly. As an example, to implement efficient internal data structures for describing floating-point numbers, `IeeeCC754++` uses integers as base data types. Since `C++` does not define the exact size of integers (e. g. **`int`** must be at least 16 bit long, but is usually 32 bit long) nor their byte ordering (endianess), `IeeeCC754++` relies on Autotools to detect the presence of the standard headers `cinttypes` and `endian.h` and set the needed definitions accordingly.

A.3 Configuring and building IeeeCC754++

All configuration to build IeeeCC754++ on/for different target user environments is handled by the configure script in the source directory. The architecture, additional FPUs, the compiler and its options are all set via configure command line options. In the following, we give a detailed description of these options.

A.3.1 Building overview

Since IeeeCC754++ uses the standard GNU build tools, the usual building semantics apply:

```
> configure [<OPTIONS>]
> make [<OPTIONS>]
> make install
```

The basic setup is done via the configure call, while make calls the compiler and further tools needed to build the binaries IeeeCC754++ and decode. If desired, make install installs these executables and the necessary test vectors into the system.

The IeeeCC754++ build system supports a wide range of options towards handling as many user environments and combinations of settings as possible. The two most important decisions before starting an actual build are the compiler that is to be used and the architecture which IeeeCC754++ is to be built for. In addition, corresponding compiler options and additional FPUs need to be chosen and specified on the command line.

After configure has finished the setup process, a summary showing details of the configured build is displayed, see Listing A.1. In this summary, all relevant details of the build are shown: whether tests should be built and hashing be used, details on the compiler and compiler flags, and the selected mode, architecture, and optional FPUs. Details are given in the next sections.

```
Build summary:
--------------

Build tests?   no
Use hashing?   yes

Compilers:     CC=gcc, CXX=g++ (g++)
32/64 bit:     native
Cross compile? no
Default flags: no

Modes:         main
Architecture:  default
FP units:      ---

Compilation flags:
    CFLAGS:
    CPPFLAGS:
```

```
        CXXFLAGS:
        LDFLAGS:
        LIBS:       -lssl -lcrypto
```

Listing A.1: *Summary output of `./configure`.*

A.3.2 Choosing an architecture

IeeeCC754++ can only be built for one architecture at a time, so this architecture must be chosen first. To generate an executable for an architecture <ARCH>, use the following syntax:

```
> configure --enable-arch-<ARCH>
```

Exactly one --enable-arch-<ARCH> parameter can be specified. If more architectures are enabled via the command line, configure emits a corresponding error message and exits.

If no architecture is explicitly selected, the default architecture is chosen (cf. Sections 3.3.3 and 5.1.1). A list of the architectures and FPUs implemented in IeeeCC754++ can be found in Table 5.1.

A.3.3 Choosing FPUs

Every architecture includes at least one FPU called the main FPU. Code for this FPU will always be built. For some architectures (like e. g. SoftFloat, see Section 5.7.2), this is sufficient to test all details of that floating-point environment. However, for most architectures, different options exist where and how to execute floating-point operations, such as x87, SSE, or AVX in x86 CPUs. To allow for testing these different FPUs with one executable, each of the n FPUs <FPU1>, ..., <FPUn> available for the chosen architecture <ARCH> can be enabled as follows:

```
> configure --enable-arch-<ARCH> --enable-fpu-<FPU1> ...
    --enable-fpu-<FPUn>
```

Furthermore, it is possible to activate all available FPUs with

```
> configure --enable-arch-<ARCH> --enable-fpu-all
```

It is important to note that for some architectures it is not possible or not advisable to build all FPUs simultaneously due to compiler restrictions or unwanted side effects. For instance, it is not advisable to build an x86 executable containing both SSE and AVX FPUs with later versions of gcc because gcc favours AVX when both SSE and AVX are requested at the same time, resulting in only AVX instructions being generated. Details are given in the respective sections in Chapter 5; for some combinations of architecture and FPUs, appropriate warnings are generated by configure.

A.3.4 Choosing the compiler and compiler options

When using Autotools as a build system, the desired compiler and its respective command line options are specified via environment variables. These can be set either directly in the execution environment (which usually consists of a shell like `bash`) or on the `configure` command line with one of the following alternatives[1]:

```
> configure ENV="<VALUES>"
> ENV="<VALUES>" configure
```

The `C++` compiler is set via the environment variable `CXX` while the `C` compiler is chosen by setting `CC`. Additionally, the `IeeeCC754++` build system supports `MYCC` and `MYCXX` for changing the compiler. If the latter variables are set, they override the values of `CC` and `CXX`.

Some architectures or FPUs need special compiler switches in order to build successfully. Some of these are set up automatically during the configuration of `IeeeCC754++`; however, it might be necessary to override the automatically chosen values or supply additional switches. Table A.2 lists the environment variables that influence the build process. It is important to note that the process of using the variables `CXXFLAGS`, `LDFLAGS`, `LIBS` etc. is different from simply specifying the desired command line options as compiler switches on a command line. This is due to the fact that most compilers act as a frontend for the toolchain of preprocessor, compiler, and linker and that options influencing only parts of these toolchain are forwarded only to the appropriate tools. As an example, switches influencing the linking step (like supplying `-lm` to link the standard math library `libm`) must be specified via `LIBS` and not via `CXXFLAGS`.

With these considerations, it is vitally important to be able to control all compiler and linker relevant settings during the setup and build process. Therefore, `configure` displays the chosen settings for the relevant environment variables in its summary at the end of the configuration step (cf. Listing A.1). An example of a build with more involved compiler settings is shown in Listing A.4, page 304.

A few more caveats and features concerning the compiler and its switches are worth discussing:

- By default, the Autotools suite uses "`-O2 -g`" as compiler switches since in most "real world" scenarios, performing moderate optimisation and including debug information makes sense. In `IeeeCC754++` however, optimisations must be regarded as potentially harmful as it is not clear in advance that floating-point semantics are not altered. Therefore, these options are deactivated in the `IeeeCC754++` build system.

[1] Note that semantically, the two alternatives differ significantly: The second version changes the environment for `configure` before execution while the first variant leaves the environment untouched, but passes the relevant variables as command line arguments. However, from a practical point of view, `configure` behaves mostly identical with both variants. The first variant is recommended; for an explanation, see the definition of "precious variables" in the Autoconf documentation [GNU16b] (search for `AC_ARG_VAR`).

Name	Use
CC	Name of the C-compiler.
CXX	Name of the C++-compiler.
MYCC	Name of the C-compiler; overrides CC.
MYCXX	Name of the C++-compiler; overrides CXX
CFLAGS	Command line options passed to the C-compiler.
CXXFLAGS	Command line options passed to the C++-compiler.
CPPFLAGS	Command line options passed to the preprocessor.
LDFLAGS	Command line options passed to the linker.
LIBS	Additional libraries; passed to the linker.

Table A.2: *Environment variables influencing the build process.*

- In some cases, the options used by the compiler might not yield the best possible floating-point results (i. e. the most IEEE-conforming floating-point results). The IeeeCC754++ build system tries to remediate this by adding compiler switches that provide better floating-point semantics for certain compilers (e. g. setting "`-fp-model strict`" for `icc`).

  However, if the default floating-point conformity for a compiler of some user environment is to be tested, it is necessary to remove these additional switches. This can be achieved with the option `--enable-default` as follows:

  ```
  > configure --enable-default
  ```

  Note that this option enables default compiler behaviour and is *not* identical to `--enable-arch-default` which selects the default architecture.

- The IeeeCC754++ build system tries to set up the build process in a way that IeeeCC754++ can be built with a minimal amount of additional effort. However, in rare cases it can be necessary to revert some of these choices and override settings with custom values. The correct values can then be supplied via the corresponding environment variables as outlined above. To get rid of the pre-chosen compiler options, use the switch `--without-compile-options`:

  ```
  > configure --without-compile-options
  ```

A.3.5 Generic build features

The IeeeCC754++ build system supports the most common standard features that the Autotools suit offers:

- Out-of-tree builds: IeeeCC754++ supports building in an arbitrary directory. In this case, `configure` must be called with appropriate path settings:

```
> mkdir <SOME_DIR>
> cd <SOME_DIR>
> <PATH_TO_IeeeCC754++>/configure
> make
> make install
```

- Installing into arbitrary directory: By default, IeeeCC754++ will install into /usr/local. To change this base path, use

  ```
  > configure --prefix=<TARGET_DIR>
  ```

- Listing all options: If a list of all options is desired, use

  ```
  > configure --help
  ```

- Parallel builds: It is possible to build several source files at once to speed up the building process. This behaviour can be achieved by using the command line option -j<BUILDS> with <BUILDS> being the number of parallel builds, e. g. use

  ```
  > configure -j8
  ```

  to build eight files at the same time.

- Verbose builds: The build system is set up in a way that only minimal output is produced during the compilation phase, i. e. it shows only the name of the file that is currently built and the program used for the build. However, it is possible to display the full command line used while building by adding "V=1" to the make command line as shown in the following (heavily shortened) example:

```
> make
Making all in src
make[1]: Entering directory '/work/diss/src/svn/build/src'
make  all-recursive
make[2]: Entering directory '/work/diss/src/svn/build/src'
Making all in common
make[3]: Entering directory '/work/diss/src/svn/build/src/common'
    ...
  CXX      IeeeCC754_classic.o
  AR       libIeeeCC754classic.a
  CXX      decode.o
  CXXLD    decode
make[3]: Leaving directory '/work/diss/src/svn/build/src/common'
    ...
make[1]: Leaving directory '/work/diss/src/svn/build'
> make clean
> make V=1
Making all in src
make[1]: Entering directory '/work/diss/src/svn/build/src'
make  all-recursive
```

```
make[2]: Entering directory '/work/diss/src/svn/build/src'
Making all in common
make[3]: Entering directory '/work/diss/src/svn/build/src/common'

    ...

g++ -DHAVE_CONFIG_H -I. -I../../../src/common -I../../src  -I../../../src
    -I../../../src/common -I../../../src/default -Wall       -MT
    IeeeCC754_classic.o -MD -MP -MF .deps/IeeeCC754_classic.Tpo -c -o
    IeeeCC754_classic.o ../../../src/common/IeeeCC754_classic.cc
mv -f .deps/IeeeCC754_classic.Tpo .deps/IeeeCC754_classic.Po
rm -f libIeeeCC754classic.a
ar cru libIeeeCC754classic.a Bitstring.o DriverFloatRepr.o FP.o FPregistry.o
    FileOps.o Hex.o IeeeCC754++_util.o Checksum.o Error.o
    IeeeCC754_classic.o
ranlib libIeeeCC754classic.a
g++ -DHAVE_CONFIG_H -I. -I../../../src/common -I../../src  -I../../../src
    -I../../../src/common -I../../../src/default -Wall       -MT decode.o
    -MD -MP -MF .deps/decode.Tpo -c -o decode.o ../../../src/common/decode.cc
mv -f .deps/decode.Tpo .deps/decode.Po
g++        -o decode decode.o libIeeeCC754++.a -lssl -lcrypto
make[3]: Leaving directory '/work/diss/src/svn/build/src/common'

    ...

make[1]: Leaving directory '/work/diss/src/svn/build'
```

A.3.6 Additional build options

IeeeCC754++ offers a few additional options to influence the configure and build process:

- The fingerprint mode and the message digests available in verbose mode (cf. Sections 3.3.4 and 3.3.6) need an underlying implementation of cryptographic message digests. In order to not unnecessarily duplicate source code and produce potentially flawed digests, IeeeCC754++ does not supply its own digest implementation, but relies on the highly renowned and widespread cryptographic library OpenSSL [SSL16]. configure detects the OpenSSL headers and libraries if they are installed in standard places. If a custom OpenSSL should be used, its location <PATH> can be specified with

  ```
  > configure --with-openssl=<PATH>
  ```

  If no OpenSSL implementation can be found, the build system compiles IeeeCC754++ without OpenSSL and thereby disables the use of the fingerprint mode and the message digests in verbose mode. If OpenSSL is present in the current environment, but building IeeeCC754++ without OpenSSL is nonetheless desired, the same effect can be achieved with

  ```
  > configure --without-openssl
  ```

- The IeeeCC754++ build system comes with a small set of utilities that check different features of the current architecture and build setup. These can be enabled with

```
> configure --with-tests
```

After compilation, the tests can be found in the subdirectory `src/tests/`. After changing into that directory, they can be executed by using the command

```
> make tests
```

Additionally,

```
> make output
```

can be used to dump all macros known by the compiler's preprocessor into the file `macros.output`. Table A.3 shows the available test programs and their purpose.

Program	Purpose
`checkBoolFloat`	Check how floating-point numbers are converted into **`bool`**s.
`checkInt`	Check the size of integer variables.
`checkTypes`	Check definitions imported from `Types.h`.
`macro`	Display macros defined by the compiler's preprocessor which are relevant inside `IeeeCC754++`.
`sanity`	Sanity check for integer variables.

Table A.3: *Test programs supplied by `IeeeCC754++`.*

- Some modern hardware platforms support more than one processor word size, e. g. the x86 architecture which commonly allows for 32-bit and 64-bit binaries to coexist and be executed in the same operating system. When configuring `IeeeCC754++`, the build system sets up the build to use the "native" word size (i. e. the kernel's word size, which is usually the larger of the supported sizes). In order to force either a 32-bit or a 64-bit build, the switches `--enable-m32` or `--enable-m64` can be used.

A.3.7 Cross compilation

Cross compilation is the process of building an executable on one platform that is intended to run on a different platform with a different ISA. In order for a cross compile build to work, a few requisites have to be met: A working cross compiler that runs on the build platform and generates code for the target platform must be installed together with the system libraries of the *target* platform that the current build will be linked against, such as the target system's `libc`, `libstdc++`, and `libm`.

The `IeeeCC754++` build system makes use of the cross compiling facilities of the Autotools family. There are basically two ways of setting up the cross build:

- The Autotools way: The correct `C` and `C++` cross compilers must be set via the `CC` and `CXX` environment variables. In addition, the switch `--host` together with the correct target triple must be supplied on the command line in order to make `configure` enter cross compilation mode. For details, see the Autotools documentation [GNU16a].
- The `IeeeCC754++` way: The target triple needed by `--host` can be supplied via the environment variable `MYHOST`. `configure` detects whether this variable is set and enters cross compile mode accordingly. The `C` and `C++` cross compilers should be set via `MYCC` and `MYCXX` in this case.

Listings A.2 and A.3 show how to set up cross compilation for both variants and some of the relevant output for an `IeeeCC754++` executable built on an x86 host to run on an ARM host with AARCH64 architecture. The cross compiler used is gcc-4.9.

```
> ../configure CC="aarch64-linux-gnu-gcc" CXX="aarch64-linux-gnu-g++"
    --enable-arch-aarch64 --host=aarch64-linux-gnu
    ...
checking build system type... x86_64-unknown-linux-gnu
checking host system type... aarch64-unknown-linux-gnu
checking target system type... aarch64-unknown-linux-gnu
checking for aarch64-linux-gnu-gcc... aarch64-linux-gnu-gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... yes
    ...
configure:

    Build summary:
    --------------

    Compilers:     CC=aarch64-linux-gnu-gcc, CXX=aarch64-linux-gnu-g++ (g++)
    32/64 bit:     native
    Cross compile? yes
    Default flags: no

    Modes:         main
    Architecture:  aarch64
    FP units:      ---
```

Listing A.2: *Setting up cross compilation via `-host`.*

```
> ../configure MYCC="aarch64-linux-gnu-gcc" MYCXX="aarch64-linux-gnu-g++"
    MYHOST="aarch64-linux-gnu" --enable-arch-aarch64
    ...
configure: Using --host=aarch64-linux-gnu from MYHOST environment variable to
    setup cross-compilation.
configure: WARNING: If a cross compiler is detected then cross compile mode will
    be used
```

```
checking build system type... x86_64-unknown-linux-gnu
checking host system type... aarch64-unknown-linux-gnu
checking target system type... aarch64-unknown-linux-gnu
configure: Using CC=aarch64-linux-gnu-gcc from MYCC environment variable.
configure: Using CXX=aarch64-linux-gnu-g++ from MYCXX environment variable.
checking for gcc... aarch64-linux-gnu-gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... yes
    ...
configure:

    Build summary:
    --------------

    Compilers:     CC=aarch64-linux-gnu-gcc, CXX=aarch64-linux-gnu-g++ (g++)
    32/64 bit:     native
    Cross compile? yes
    Default flags: no

    Modes:         main
    Architecture:  aarch64
    FP units:      ---
```

Listing A.3: *Setting up cross compilation via* `MYHOST`.

A.3.8 Building historic modes

The original `IeeeCC754` implementation differentiated between executables for the basic operations and for conversions (cf. Section 1.2.3). For most user environments, this distinction purely reflected a systematic point of view. The implementations for the x86 platform however (called IntelPentium and AMD in `IeeeCC754`) made use of the two executables by supplying two implementations of the remainder and square root operations: the "basic" executable includes inline assembler versions of these operations while the "conversion" executable calls the standard `C` families of functions `sqrtf`, `sqrt`, `sqrtl`, and `remainderf`, `remainder`, `remainderl`, respectively.

In `IeeeCC754++`, this distinction is no longer necessary due to the concept of FPUs which would separate the two versions of these functions into different FPU implementations. For historic reasons, the x86 architecture can still be built with basic and conversion executables (in addition to the "standard" executable). This build behaviour is achieved by using the `--enable-mode-<MODE>` options shown in table Table A.4

Note that for each mode given a separate executable is built. In the basic and conversion mode executables the floating-point operators are implemented in the `main` FPU and will be executed by default (i. e. without specifying additional command line switches).

Option	Effect
`--enable-mode-main`	Build a “main” mode executable; default setting.
`--enable-mode-basic`	Build a “basic” mode executable.
`--enable-mode-conv`	Build a “conversion” mode executable.
`--enable-mode-all`	Build all three modes.

Table A.4: *Environment variables influencing the build process.*

If no `--enable-mode-<MODE>` switch is explicitly given, only the “main” mode executable will be built.

A.3.9 A detailed example

Listing A.4 gives a detailed example of building `IeeeCC754++` for the x86 architecture, together with the x87 and SSE FPUs. The executable is compiled with `gcc-4.8` with the additional option `-g` that includes debug information into the resulting binary. Some additional tests are enabled via `--with-tests`.

```
> ../configure CC=gcc-4.8 CXX=g++-4.8 CXXFLAGS='g' --enable-arch-x86
    --enable-fpu-x87 --enable-fpu-sse --with-tests
    ...
configure:

    Build summary:
    --------------

    Build tests?   yes
    Use hashing?   yes

    Compilers:     CC=gcc-4.8, CXX=g++-4.8 (g++)
    32/64 bit:     native
    Cross compile? no
    Default flags: no

    Modes:         main
    Architecture:  x86
    FP units:      x87 sse

    Compilation flags:
        CFLAGS:
        CPPFLAGS:
        CXXFLAGS:  -g -msse -msse2
        LDFLAGS:
        LIBS:      -lssl -lcrypto


> make
    ...
Making all in x86
make[3]: Entering directory '/work/diss/src/svn/bbb/src/x86'
  CXX      fpenv_x86.o
```

```
  CXX      myfenv.o
  CXX      fpu_x87.o
  CXX      fpu_sse.o
  CXX      fpenv_sse.o
  CXX      main_x86.o
  CXX      fpu_main.o
  CXXLD    IeeeCC754++_x86
make[3]: Leaving directory '/work/diss/src/svn/bbb/src/x86'
    ...
> ./src/x86/IeeeCC754++_x86 -vio alls -f output.log --fpu=sse
```

Listing A.4: *Example build for the x86 architecture.*

The build summary shows that the OpenSSL libraries installed on the test system were detected and the generation of message digests activated accordingly. Furthermore, the additional compiler switches `-msse` and `-msse2` have been added in order to enable building the SSE FPU.

The compilation is started by issuing `make`; only the output of the architecture build is shown. Afterwards, `IeeeCC754++` is not installed into the system, but the compiled `IeeeCC754++` binary executed directly. Note that it is located in the directory `./src/<ARCH>` and called `IeeeCC754++_<ARCH>`. Also note that `alls` describes a hypothetical test vector file located in the current directory, and output for the testing process of the SSE FPU is written to `output.log`.

Appendix B

Adding a new architecture to IeeeCC754++

One major advantage of setting up the IeeeCC754++ build system centred around the concepts *architectures* and *FPUs*, as well as restructuring the code basis and file layout to support these concepts (cf. Section 3.1.5 and Appendix A), lies in the (relatively) easy extensibility of IeeeCC754++ with code supporting a new floating-point environment. In this chapter, we discuss all steps necessary to extend IeeeCC754++ with an architecture xyz containing the mandatory main FPU and an additional FPU called abc. Adding further FPUs follows the same process.

In order to simplify the extension process, the IeeeCC754++ source tree includes an implementation of an architecture called dummy containing all relevant parts of an architecture implementation (i. e. inheriting from the right classes, setting up internal data, etc.) without code to actually perform floating-point operations. Furthermore, the dummy architecture contains an FPU called generic in order to show the additional steps necessary to add further FPUs to a new architecture. The corresponding dummy source files are heavily commented, as well as the file configure.ac to which the new architecture has to be added in order to enable building it.

In the following, we give a step by step description of the extension process to add the new architecture xyz together with its FPU abc.

B.1 File structure

The first step is to create a new directory in src/ that will contain all files needed for the architecture. Although in principle an arbitrary name can be chosen for this directory, it is advisable to use the name of the new architecture:

```
> mkdir src/xyz
```

The easiest way to set up all needed source files for the new architecture and FPU is to copy (and later modify) the corresponding files from the `dummy` architecture. We start by copying the Automake file necessary for generating an appropriate `Makefile`:

```
> cp src/dummy/Makefile.am src/xyz/
```

The code for the `dummy` architecture and its `main` FPU can be copied as they are:

```
> cp src/dummy/DriverFloat_main.h src/xyz/
> cp src/dummy/fpu_main.cc src/xyz/
```

The source files for the new FPU `abc` can be copied from the `generic` FPU, but need to be renamed:

```
> cp src/dummy/DriverFloat_generic.h src/xyz/DriverFloat_abc.h
> cp src/dummy/fpu_generic.cc src/xyz/fpu_abc.cc
```

Finally, a source file containing the `main()` function needs to be added:

```
> cp src/dummy/main_dummy.cc src/xyz/main_xyz.cc
```

These files constitute the minimum setup for a new architecture with an additional FPU. If further code is necessary in order to initialise the FPU that is to be tested or any other source code, it can be added to these existing files or may be placed in separate files, preferably inside the directory `src/xyz/`. Note that if these additional files need to be compiled, the build system must be given knowledge of these files by adding them to `Makefile.am` (see Section B.3).

If no additional FPU `abc` is needed, the corresponding source files are unnecessary and need not be copied.

B.2 Build system: `configure.ac`

After setting up the files for the new architecture, the build system must be given knowledge how to build the code for the new implementation. To achieve this, the file `configure.ac` needs to be modified. The places where code has to be added are commented and marked with "`STEP` n", so doing a text search for "`STEP` n" yields the correct locations. Steps that are necessary only when a new FPU is added are marked "*(optional)*".

[STEP 1] Since building code for an architecture `xyz` is selected by calling `configure` with the option `--enable-arch-xyz`, this switch needs to be added to `configure.ac`:

```
AC_ARG_ENABLE([arch-xyz],
  [AS_HELP_STRING([--enable-arch-xyz],
    [Set ARCH to XYZ.])],
    [],
    [enable_arch_xyz=no])
AM_CONDITIONAL([ARCH_XYZ], [test x$enable_arch_xyz = xyes])
```

This code fragment enables the mentioned command line switch and sets up an Automake variable `ARCH_XYZ` needed in later `Makefile.am` files.

Note that the string "`Set ARCH to XYZ`" in line three depicts a comment that is displayed by `configure -help`. As such, its contents should be chosen in a manner that it clearly explains the meaning of the corresponding option does.

[STEP 2] *(optional)* The same process is needed to set up the build system for building the FPU `abc`:

```
AC_ARG_ENABLE([fpu-abc],
  [AS_HELP_STRING([--enable-fpu-abc],
    [Enable abc FPU instruction set for arch xyz.])],
    [],
    [enable_fpu_abc=no])
```

Similar to step 1, the third line consists of a command for `configure`'s built in help system.

[STEP 3] Here, code specific to the new architecture can be added, e. g. extra compiler options needed to compile successfully or additional environment settings. The first two lines inside the `if` statement are necessary for the build system to work; after these lines, custom code can be added. Examples can be found in `configure.ac` for the architectures already implemented in `IeeeCC754++`.

```
if test x$enable_arch_xyz = xyes
then
  MYARCH="xyz"
  archs=‘expr $archs + 1‘

  # place custom code here

fi
```

[STEP 4] *(optional)* For each additional FPU that will be implemented (such as `abc` in this example), a variable is needed to describe whether these FPUs should be built or not. It should be initialised with the value "`no`":

```
mh_fpu_abc=no
```

[STEP 5] *(optional)* Furthermore, for every new FPU code is necessary to ensures that the build system enables the building of this FPU. Also in this step, custom setup can be added in case further compiler options or environment variables are needed to build code for the new FPU.

```
if test x$enable_arch_xyz = xyes
then
  if test x$enable_fpu_abc = xyes -o x$enable_fpu_all = xyes
  then
    mh_fpu_abc=yes
    FPU="${FPU}abc "
    AC_DEFINE([BUILD_FPU_ABC], [1], [Build abc FPU code.])

    # place custom code here

  fi
fi
```

First, this code fragment ensures that the FPU `abc` is only enabled when `IeeeCC754++` is compiled for the architecture `xyz`. Then, the FPU is enabled if it is explicitly chosen or if all FPUs for the current architecture should be built (i. e. when the `configure` option `--enable-fpu-all` was used). Afterwards, the FPU to be built is added to the text list "`FPU`", and the Autoconf variable `BUILD_FPU_ABC` is set which enables building the FPU.

[STEP 6] *(optional)* In addition to the Autoconf variable `BUILD_FPU_ABC`, an Automake variable `FPU_ABC` is necessary to ensure that the FPU `abc` is built:

```
AM_CONDITIONAL([FPU_ABC], [test x$mh_fpu_abc = xyes])
```

[STEP 7] *(optional)* The last step of handling the new FPU `abc` consists of verifying that we only build this FPU when the build is configured for the corresponding architecture (in this case `xyz`):

```
AS_IF([test x$enable_fpu_abc = xyes], AS_IF([test x$mh_fpu_abc !=
    xyes],
    AC_MSG_WARN([--enable-fpu-abc is not supported on "${MYARCH}"
        arch!]), []), [])
```

[STEP 8] Finally, the build system needs to be told that an appropriate `Makefile` needs to be generated for the architecture `xyz`. This can be achieved by adding its full path to the Autoconf variable `AC_CONFIG_FILES` as follows:

```
AC_CONFIG_FILES([
  Makefile
  src/Makefile
  src/common/Makefile
    ...
  src/dummy/Makefile
```

```
  src/xyz/Makefile
])
```

B.3 Build system: Makefile.am

At this point, the source tree contains all necessary files for the new architecture and the new FPU, and the build system contains all knowledge and command line options necessary to configure a build of the architecture `xyz` and the FPU `abc`. In the next step, the actual compilation of the newly added source files needs to be set up by modifying the Automake file `src/xyz/Makefile.am`:

- The file `Makefile.am` copied from the `dummy` architecture contains code to build an FPU `generic`. This code should be modified as follows to enable building the FPU `abc`:

```
## add files for abc FPU
## can be commented when no FPU is needed
if FPU_ABC
  fpuheaders += DriverFloat_abc.h
  fpusources += fpu_abc.cc
endif
```

  For additional FPUs, this code should be copied and modified accordingly. If no additional FPU is needed, the respective lines can either be left as they are[1] or commented out.

- If additional source files were added inside `src/xyz/` which need to be compiled, they should be added to the variables `archheaders` and `archsources` as follows (in this example, we add a new header file called `newheader.h` and a source file `newheader.cc`):

```
## header and src files for xyz arch
archheaders = DriverFloat_main.h newheader.h
archsources = fpu_main.cc newheader.cc
```

B.4 Implementing the new architecture

Since `IeeeCC754++`'s purpose is to check the characteristics of a given floating-point environment, the base "ingredient" for a new architecture is an underlying FPU

[1] This does no harm as the build system has no knowledge about a `generic` FPU for the new architecture `xyz` and will therefore not set the variable `FPU_GENERIC` to true.

implementation. This FPU is called the `main` FPU and contained in every architecture implemented in `IeeeCC754++`. Consequently, adding a new architecture means implementing the main FPU for this architecture.

Inside the `IeeeCC754++` code base, all FPU-dependent features are implemented in classes inheriting from `DriverFloatRepr` (see Figure A.2). The class `DriverFloatRepr` itself contains an empty FPU implementation so that in all inherited classes, only those methods that exist in the new FPU need to be overloaded. Additionally, some constructors and conversion operators need to be implemented. In the following, we give a detailed description of the steps necessary to add a working `main` FPU for the new architecture `xyz` by implementing the class `DriverFloat_main`.

B.4.1 `src/xyz/DriverFloat_main.h`

The definition of the class `DriverFloat_main` is contained in the source file `src/xyz/DriverFloat_main.h`. Since it was copied over from the `dummy` architecture, it contains definitions of some constructors and conversion operators, some (necessary) helper functions, and all operators known to `IeeeCC754++`. Only the definitions of those operators actually going to be implemented are necessary, so all other operator definitions can be commented (or deleted).

- The basic class layout with default constructors looks as follows:

```
class DriverFloat_main: public DriverFloatRepr
{
  public:
    DriverFloat_main() : DriverFloatRepr() { }
    DriverFloat_main(int m, int e, int h) : DriverFloatRepr(m, e, h) { }
    DriverFloat_main(Bitstring & fp, int m, int e, int h) :
        DriverFloatRepr(fp, m, e, h) { }
    DriverFloat_main(const DriverFloat_main & r) : DriverFloatRepr(r) { }

    DriverFloat_main(const DriverFloatRepr & r) : DriverFloatRepr(r) { }
};
```

  In general, further class attributes in addition to those which are defined in `DriverFloatRepr` should not be necessary, so the first four constructors initialise the class `DriverFloat_main` by forwarding to the base class constructors. The fifth constructor is needed when operators of the base class should be called. In general, this should not be necessary for the `main` FPU as the base class consists of only empty implementations.

- After the execution of a test vector, the result of that operation needs to be converted from a floating-point number or integer into `IeeeCC754++`'s internal representation, i. e. into an instance of type `DriverFloat_main`. This is achieved with the following conversion constructors:

```
public:
  DriverFloat_main(float);
  DriverFloat_main(double);
  DriverFloat_main(long double);

  DriverFloat_main(int32_t);
  DriverFloat_main(uint32_t);
  DriverFloat_main(int64_t);
  DriverFloat_main(uint64_t);
```

- In order to execute floating-point operations on the target user environment, the floating-point numbers in the current test vector need to be converted into the corresponding C++ data types **float**, **double** etc. This is achieved by implementing the following conversion operators:

```
public:
  float tofloat();
  double todouble();
  long double tolongdouble();

  int32_t toint();
  uint32_t touint();
  int64_t toint64();
  uint64_t touint64();
```

- In order to set the rounding mode for the test vector under investigation and to check for the correct application of exception flags, methods are necessary that handle the floating-point environment. The signature of the necessary methods is as follows:

```
protected:
  virtual void SetLibRound();
  virtual void ClearLibExceptions();
  virtual void GetLibExceptions();
```

`SetLibRound()` sets the rounding mode while all currently set exceptions flags are cleared via `ClearLibExceptions()`. These two methods must be called before a vector is executed in order to ensure the operation is started in a clean floating-point environment. However, instead of having to call both of these methods explicitly every time an operation is to be tested (which is crucial to get meaningful testing results), they are implicitly called inside the method `DriverFloatRepr::SetLibEnvironment()` which also does some additional set up of vector FPU units (see Section B.7). As a consequence, after setting up the operands to be tested, `SetLibEnvironment()` must be called before executing the test operation.

After executing a test, another method is necessary to retrieve the newly set floating-point exception flags (if any): the method `GetLibExceptions()`

must be overloaded in `DriverFloat_main` and be explicitly called after every test vector execution.

All environment information needed to set up the current test vector is stored inside that test vector itself, i. e. in the current instance of `DriverFloat_main`. In order to retrieve the current rounding mode and to return exceptions, the following functions can be used:

```
uint32_t GetFPRound();
void SetFPDivByZero();
void SetFPInvalid();
void SetFPUnderflow();
void SetFPOverflow();
void SetFPInexact();
```

`GetFPRound()` returns the current rounding mode, i. e. one of the values `RM_NEAR`, `RM_AWAY`, `RM_ZERO`, `RM_UP`, or `RM_DOWN`, whereas the other functions set the corresponding exception flags. Note that it is only necessary to use the latter functions inside `GetLibExceptions()` when floating-point exceptions were encountered during the execution of the test vector. For further details on using these functions, see e. g. `src/default/fpenv_default.cc` or `src/softfloat/fpenv_softfloat.cc`.

- All operations which are supported in the architecture `xyz`, and that therefore should be tested inside the `main` FPU, need to be defined inside the class `DriverFloat_main`. The following example shows the signatures of a short selection of some important operators; for a full list of supported operations, see Table B.1.

```
DriverFloat_main operator + (DriverFloat_main &);
DriverFloat_main operator - (DriverFloat_main &);
DriverFloat_main operator * (DriverFloat_main &);
DriverFloat_main operator / (DriverFloat_main &);
DriverFloat_main operator % (DriverFloat_main &);
DriverFloat_main sqrt();
DriverFloat_main fma(DriverFloat_main &, DriverFloat_main &);
```

For the implementation of the basic operators, `C++`'s operator overloading features are used so e. g. adding two floating-point values `x` and `y` of type `DriverFloatRepr` can be performed by writing `x + y`. The operation `sqrt()` is an example of an operation with only one operand (which is already stored in the current variable of type `DriverFloat_main`), whereas `fma()` is currently the only operator in `IeeeCC754++` that takes three operands.

- As outlined in Section B.8, it is possible to specify which operators and rounding modes should be tested inside the current FPU by registering them into a (static) variable of type `FPregistry`. This process, as well as setting up the FPU for operations on SIMD vectors, is done by overloading the method `Register()`:

```
public:
  void Register();
```

B.4.2 src/xyz/fpu_main.cc

The methods defined in the class `DriverFloat_main` are implemented inside the file `src/xyz/fpu_main.cc`. In this section, we outline this implementation for selected methods; see also Section B.6 for further details.

- In order to implement the three functions that handle the platform's floating-point environment (here, environment means rounding mode, intermediate floating-point formats etc.), it must be known how setting rounding modes and setting or retrieving exception flags are handled on that platform. The more common architectures (or, more precisely, their operating systems) support the `C` family of floating-point environment handling functions such as `fesetround()`, `feclearexcept()`, and `fetestexcept()`, so these can be directly used. An example implementation employing these functions can be found in `src/default/fpenv_default.cc`.

  However, when these standard `C` functions are not supported, custom code has to be written to handle rounding modes and exception flags. Examples of custom implementations can be found in `src/mpfr/fpenv_mpfr.cc` or `src/softfloat/fpenv_softfloat.cc`.

- In all floating-point operators that will be implemented in the new architecture, the floating-point values in the current test vector need to be converted into `C++` data types, and the result must be converted back into the type `DriverFloat_main`. The code of the corresponding conversion operators and constructors is obviously user environment dependent; however in practice, the main difference between (hardware) architectures consists of the endianess layout. The `default` architecture includes an implementation supporting both big and little endian layouts (see `src/default/fpenv_default.cc`), whereas examples for only big and little endian layouts can be found in the x86 and Blue Gene/Q architectures (i. e. in the files `src/x86/fpenv_x86.cc` and `src/bgq/fpenv_bgq.cc`, respectively).

- In order to specify which rounding modes and operations are to actually be tested inside the new architecture, these must be registered into the internal registry (implemented as the static variable `DriverFloatRepr::FPreg`). This registration process must be done in `Register()`, for details cf. Section B.8. The following short example shows how to enable all rounding modes and operations:

```
void DriverFloat_main::Register()
```

```
{
    FPreg.clearall();
    FPreg.registerRDall();
    FPreg.registerOPall();
}
```

- Finally, the actual operators need to be implemented. A detailed description of the steps necessary to set up the involved floating-point variables and the floating-point environment is given in Section B.6.

B.5 Adding an FPU

The process of adding an additional FPU to a new architecture is largely identical to the process of adding the architecture's `main` FPU. The main difference lies in the fact that for a new FPU `abc`, the corresponding class `DriverFloat_abc` is not inherited from `DriverFloatRepr`, but from `DriverFloat_main` and in general can re-use at least some of the methods implemented inside `DriverFloat_main`. In most cases, it is e. g. possible to use the different conversion operations between `DriverFloat_abc` and `C++`'s data types already defined in `DriverFloat_main` instead of implementing custom variants.

The methods that obviously need to be overloaded inside `DriverFloat_abc` are the floating-point operators, the `Register()` method and the constructors (by forwarding these to the base class operators). To arrive at a minimal implementation of the FPU `abc`, based on the `generic` FPU from the `dummy` architecture, all occurrences of "`DriverFloat_generic`" in the file `src/xyz/DriverFloat_abc.h` (which was copied from `src/dummy/DriverFloat_generic.h`) need to be replaced by "`DriverFloat_abc`". The resulting code is shown below (slightly shortened and with modified comments):

```
#ifndef _DRIVERFLOAT_ABC_H
#define _DRIVERFLOAT_ABC_H

#include <Types.h>
#include <DriverFloat_main.h>

class DriverFloat_abc: public DriverFloat_main
{
  public:
    void Register();

    // basic constructors
    DriverFloat_abc() : DriverFloat_main() { }
    DriverFloat_abc(int m, int e, int h) : DriverFloat_main(m, e, h) { }
    DriverFloat_abc(Bitstring & fp, int m, int e, int h) : DriverFloat_main(fp,
        m, e, h) { }
    DriverFloat_abc(const DriverFloat_abc & r) : DriverFloat_main(r) { }

    // conversion constructors
    DriverFloat_abc(float f) : DriverFloat_main(f) { }
```

```
    DriverFloat_abc(double d) : DriverFloat_main(d) { }
    DriverFloat_abc(long double l) : DriverFloat_main(l) { }
    DriverFloat_abc(int32_t i) : DriverFloat_main(i) { }
    DriverFloat_abc(uint32_t u) : DriverFloat_main(u) { }
    DriverFloat_abc(int64_t l) : DriverFloat_main(l) { }
    DriverFloat_abc(uint64_t ul) : DriverFloat_main(ul) { }

    // constructors needed to use methods from the base classes
    // that have DriverFloatRepr or DriverFloat_main as operands
    DriverFloat_abc(const DriverFloat_main & r) : DriverFloat_main(r) { }
    DriverFloat_abc(const DriverFloatRepr & r) : DriverFloat_main(r) { }

};
#endif // _DRIVERFLOAT_ABC_H
```

All constructors simply forward to their base class equivalents. This example omits all floating-point operators which must be added to `DriverFloat_abc` and implemented in the file `fpu_abc.cc`.

B.6 Implementing an operation

When implementing a floating-point operation inside a new FPU, the following steps are always necessary in order to set up the FPU for execution of that operation and enabling `IeeeCC754++` to evaluate the result of the operation:

- Converting the operands into the FPU's binary format.
- Initialising the floating-point environment.
- Executing the operation.
- Retrieving floating-point exception information from the environment.
- Converting the result into `IeeeCC754++`'s internal floating-point format.

In the following, we describe in detail how these steps can be implemented in the new FPU. In this guide, we implement the addition operation for single and double operands for the `main` FPU of the architecture `xyz`. This method has the following signature:

```
DriverFloat_main DriverFloat_main::operator + (DriverFloat_main & m);
```

[STEP 1] The first step is to differentiate between the possible source and target floating-point formats. To help detecting the current floating-point format, `IeeeCC754++` supplies the methods `isIEEEbinary16()`, `isIEEEbinary32()`, `isIEEEbinary64()`, `isIEEEbinary80()`, and `isIEEEbinary128()` that can be used to detect IEEE-conforming half, single, double, extended-double, and quadruple precision formats. If other formats are possible in the FPU

implementation and need to be recognised, the values of FP::sizeExp, FP::sizeMant, and FP::hidden need explicitly be checked. The following two variants of testing for the double format are equivalent:

```
if (isIEEEbinary64()) { ... }
if (sizeExp == 11 && sizeMant == 52 && hidden) { ... }
```

For the rest of this example, we only show code for operands in the single format; see Section B.9 for an implementation for double (which is largely identical).

Note that IeeeCC754++ also supplies methods that check for the format of the target floating-point numbers for the conversion operators, i. e. for the method isIEEEbinary32(), an equivalent method disIEEEbinary32(**int** dsizeExp, **int** dsizeMant, **int** dhidden) exists that checks for single precision target format (with "d" denoting "destination").

[STEP 2] Second, variables of the FPU's floating-point data type that store the operands and the result need to be declared. In this example, this data type is simply the C++ equivalent for the single format which is called **float**.

```
        float res, op1, op2;
```

[STEP 3] Afterwards, the floating-point values stored in ***this** and m (cf. the method's signature shown above) must be converted into the FPU's floating-point format with the method tofloat():

```
        op1 = tofloat();
        op2 = m.tofloat();
```

Note that for SIMD vector FPUs, appropriate vector operands have to be set up. For details on available helper funtions, see Section B.7; an example implementation for a vector operation is given for the abc FPU in Section B.9.

[STEP 4] Before executing the operation, the floating-point environment needs to be properly set up, i. e. the rounding mode needs to be set and the floating-point exceptions flags cleared via SetLibEnvironment, see above:

```
        SetLibEnvironment();
```

[STEP 5] At this point, everything has been prepared for the operation to be executed, so the appropriate function in the FPU can be called. In this simple example, we simply rely on the compiler to generate code for the fictitious abc FPU by using the C++ addition operator for **float**s:

```
res = op1 + op2;
```

[STEP 5] After the operation has been retrieved, the floating-point environment needs to be checked to see if (and which) exception flags have been set:

```
GetLibExceptions();
```

[STEP 6] *(optional)* When the FPU is a SIMD vector unit, an additional step is necessary to ensure the FPU operates identically on all entries inside the operand vectors. This can be conveniently done by using corresponding helper functions from `Vector.h` (see Section B.7) and setting a vector error if needed. Here, `res` denotes the result vector and `n` the number of values contained in `res`:

```
if (checkVec(res, n))
{
    setVectorError();
}
```

Similar facilities exist when checking scalar variants of a vector FPU (cf. Section 3.1.12):

```
if (checkVecScalar(res, n))
{
    setScalarError();
}
```

An example for the implementation of a vector operation can be found in Section B.9.

[STEP 7] Finally, the returned result needs to be converted into `IeeeCC754++`'s internal floating-point format and passed as return value of this method:

```
DriverFloat_main r(res);
return r;
```

The conversion is necessary to enable `IeeeCC754++` comparing the returned result with the correct result.

[STEP 8] After code has been added for every floating-point format that is supported by the FPU following steps 1 to 6, it is necessary to tell `IeeeCC754++` what to do when a floating-point format not known by the implementation of the current operator is encountered. This is achieved by returning a special value that tells `IeeeCC754++` to ignore the results of the execution of the current method:

```
    // for every other case, return a dummy DriverFloatRepr that
    // denotes that evaluation of the result should be skipped.
    return skipTest();
```

The full example implementation of the addition operator for the `main` FPU supporting single and double operands is shown in Listing B.1.

```
// example implementation for addition
DriverFloat_main DriverFloat_main::operator + (DriverFloat_main & m)
{
    // single/binary32 operands
    if (isIEEEbinary32())
    {
        // variables needed for result and operands
        float res, op1, op2;

        // convert operands from DriverFloatRepr (IeeeCC754++'s internal
        // FP format) to format of the FP implementation
        op1 = tofloat();
        op2 = m.tofloat();

        // set rounding mode etc., execute operation, retrieve exceptions
        SetLibEnvironment();
        res = op1 + op2;
        GetLibExceptions();

        // generate DriverFloatRepr from result and return it
        DriverFloat_main r(res);
        return r;
    }
    // double/binary64 operands
    else if (isIEEEbinary64())
    {
        // variables needed for result and operands
        double dres, dop1, dop2;

        // convert operands from DriverFloatRepr (IeeeCC754++'s internal
        // FP format) to format of the FP implementation
        dop1 = todouble();
        dop2 = m.todouble();

        // set rounding mode etc., execute operation, retrieve exceptions
        SetLibEnvironment();
        dres = dop1 + dop2;
        GetLibExceptions();

        // generate DriverFloatRepr from result and return it
        DriverFloat_main rd(dres);
        return rd;
    }

    // for every other case, return a dummy DriverFloatRepr that
    // denotes that evaluation of the result should be skipped.
    return skipTest();
}
```

Listing B.1: *Implementation of addition for* `DriverFloat_main`.

B.7 Handling vector FPUs

When an FPU operates on some variant of SIMD vectors (i. e. the operands consist of several floating-point values operated on at the same time with the same operation), `IeeeCC754++` must be told that it is handling a vector unit in order to enable its vector handling facilities (especially checking whether the FPU handles all entries in a vector in an identical way and setting a vector error if differences are detected). Furthermore, the handling of data types inside the methods implementing the floating-point operations is different. For both cases, `IeeeCC754++` contains facilities to help setting up the relevant data. Initialising the FPU as a vector unit is described in Section B.8.2. Here, we describe the methods available to enable convenient handling of vector data types.

`IeeeCC754++`'s philosophy of checking vector FPUs is as follows: The vector operands are filled with identical copies of the current operation's operands. Afterwards, the (vector) operation is executed, and in addition to checking the floating-point environment and the returned result (i. e. whether the returned floating-point value is correct), it is checked whether all values in the result vector are identical. For further details and the corresponding scalar check, see Section 3.1.12.

Note that to test a vector unit, a variety of approaches filling the vector with floating-point values is possible: using the same value in all positions (as done in `IeeeCC754++`), writing different values to the different positions to avoid bit-bias, or using several calls to the vector unit while putting the value into each possible position (and setting all other positions to zero). The actual testing strategy is chosen in the implementation of the operations inside a subclass of `DriverFloatRepr`, where especially the third approach could be useful for certain FPUs. The second approach of using different floating-point values in all positions however faces some issues: In general, it is only possible to set one rounding mode before executing a vector operation, and it is usually not possible to retrieve exception flags for the positions individually. Furthermore, for technical reasons, only one set of operands is known to `IeeeCC754++` when the vector operation is executed. `IeeeCC754++` employs the first strategy of filling the test vector with identical values in order to stress the full vector unit while being able to easily handle setting the rounding mode and retrieval of exceptions.

In order to make use of `IeeeCC754++`'s vector helper functions, the file `Vector.h` needs to be included. It contains three types of helper functions:

- Functions that copy a floating-point value to all slots of a vector data type.
- Functions that check if all values contained inside a vector are identical (and similar functions for checking scalar variants).
- Functions to convert vectors to a different vector format and to print a vector for debugging purposes.

The first category consists of the following functions:

```
template <typename fp, int veclen> void fillVec(fp * dst, const fp & src);
template <typename fp> void fillVec(fp * dst, fp src, const int veclen);
template <typename fp> void fillVecScalar(fp * dst, fp src, const int veclen);
template <typename fp> void fillVecHorizontal(fp * dst, fp src1, fp src2, const
    int veclen);
```

`fillVec()` copies the floating-point value `src` into the destination vector `dst` that contains `veclen` elements. Two variants of this functions exist to work around limitations of some compilers (namely NVidia's `nvcc`): One takes `veclen` as a template parameter, in the other, `veclen` is passed as a function argument. `fillVecScalar()` copies `src` into the first slot of `dst` and fills the remaining `veclen` -1 slots with the value 0. The function `fillVecHorizontal()` is a special variant needed for some SSE variants in the x86 architecture (see Section 5.2.1): It copies the two values `src1` and `src2` alternatingly into `dst`, each of them $\frac{\texttt{veclen}}{2}$ times.

```
template <typename fp, int veclen> bool checkVec(const fp * src);
template <typename fp> bool checkVec(const fp * src, const int veclen);
template <typename fp, int veclen> bool checkVecBits(const fp * src);
template <typename fp> bool checkVecBits(const fp * src, const int veclen);
template <typename fp> bool checkVecScalar(const fp * src, const int veclen);
template <typename fp> bool checkVecStride(const fp * src, const int veclen, int
    stride = 2);
```

The second family of functions consists of different variants to check whether the passed vector contains the expected values. In case of the functions `checkVec()` and `checkVecBits()`, it is verified that all `veclen` slots in the vector `src` are filled with the same value; the first variant by using the comparison operator for the type `typename fp`, the second using bitwise comparison.

`checkVecScalar()` performs checks only on all entries except the first: It performs a bitwise comparison of the entries to ensure they are identical and additionally checks if the second entry (and therefore all other values if they are identical) is equal to 0. `checkVecStride()` is another special variant needed for the x86 SSE FPU: It verifies that the entries at the positions $i \equiv p \bmod \texttt{stride}$ with $i = 0, \ldots, \texttt{stride}$ and $p = 0, \ldots, \texttt{veclen}$ are identical.

```
template <typename fpin, typename fpout> void convertVec(fpout * dst, fpin *
    src, const int veclen);
template <typename fp> void printVec(const fp * src, const int veclen);
```

Finally, `convertVec()` takes two vectors of identical length and copies all entries from the source to the target vector whereas `printVec` prints all values in the vector `src` onto the terminal.

In Section B.9, an example is shown of a vector FPU and an example implementation of the `sqrt()` operation.

B.8 Initialising an FPU

B.8.1 Registering operations and rounding modes

After implementing the functions necessary to handle the floating-point environment and all operators that should be tested inside the FPU, it is necessary to tell `IeeeCC754++` which operations and rounding modes should actually be checked. By default, no operations and rounding modes are registered for testing, resulting in empty result files due to no testing taking place.

The registration system is designed to be as flexible as possible. It is handled via the static attribute `FPregistry FPreg` of the class `DriverFloatRepr`. Every operation and rounding mode can be registered and unregistered into or from this central registry, and several helper functions exist to register e. g. all operations and rounding modes.

One feature of the design is crucial to understand: There are actually two ways to prevent a test vector from being executed and tested. First, it is possible to exclude all test vectors for some operation by not registering this operation in `FPreg`. The second variant enables exclusion even if the desired operation is implemented inside the FPU: When the function returns a special value via the method `skipVector()` (see above), the results of this single test vector are discarded, and checking is skipped. This method is used inside the file `src/common/DriverFloatRepr.h` for all operations available in `IeeeCC754++` (which are all defined inside `DriverFloatRepr`): They only contain a call to `skipVector()`, thus preventing results from being generated (since no floating-point operations are executed inside the operator) and from being evaluated (via the skip mechanism).

As a consequence, when only the desired operators supported by the FPU are implemented, it is safe to register all operations into `FPreg` as test vectors, since operations for which no implementation exists are skipped via the second mechanism just described.

The registration process is performed by implementing the corresponding calls to `FPregistry` methods inside the method `Register()` which is called automatically by the internal initialisation methods (like constructors) of `DriverFloatRepr`. The methods implemented in the class `FPregistry` take operations and rounding modes as parameters. These are implemented inside the classes `OP` and `RD` (cf. Figure A.2, page 291); the possible values can be found in Tables B.1 and B.2

Name	Operation	`OP` name	Signature
Addition	$x + y$	`OP::add`	`T T::operator + (T & y)`
Subtraction	$x - y$	`OP::sub`	`T T::operator - (T & y)`
Multiplication	$x * y$	`OP::mul`	`T T::operator * (T & y)`
Division	x/y	`OP::div`	`T T::operator / (T & y)`
Remainder	$x \bmod y$	`OP::rem`	`T T::operator % (T & y)`

Continued on next page...

Name	Operation	OP name	Signature
Square root	$\sqrt{x}$	`OP::sqrt`	`T T::sqrt()`
`fma`	$x * y + z$	`OP::fma`	`T T::fma(T & y, T & z)`
Round to		`OP::rt`	`T T::rt(T & y)`
Convert to		`OP::ct`	`T T::ct(T & y)`
Integral value		`OP::i`	`T T::rint(T & y)`
fp ⇒ int		`OP::ri`	`T T::ri()`
		`OP::ru`	`T T::ru()`
		`OP::rI`	`T T::rI()`
		`OP::rU`	`T T::rU()`
int ⇒ fp		`OP::ci`	`T T::ci(int e, int m, int h)`
		`OP::cu`	`T T::cu(int e, int m, int h)`
		`OP::cI`	`T T::cI(int e, int m, int h)`
		`OP::cU`	`T T::cU(int e, int m, int h)`
bin ⇔ dec	$x_2 \Rightarrow x_{10}$	`OP::b2d`	`T T::b2d(int p)`
	$x_{10} \Rightarrow x_2$	`OP::d2b`	`T T::d2b()`
Roots	$\sqrt[3]{x}$	`OP::cbrt`	`T T::cbrt()`
	$1/\sqrt{x}$	`OP::rsqrt`	`T T::rsqrt()`
	$\sqrt[n]{x}$	`OP::rootn`	`T T::rootn(T & y)`
Power	x^y	`OP::pow`	`T T::pow(T & y)`
	x^n	`OP::pown`	`T T::pown(T & y)`
	$x^y, x > 0$	`OP::powr`	`T T::powr(T & y)`
Trigonometric	$\sin(x)$	`OP::sin`	`T T::sin()`
functions	$\cos(x)$	`OP::cos`	`T T::cos()`
	$\tan(x)$	`OP::tan`	`T T::tan()`
	$\text{atan2}(y, x)$	`OP::atan2`	`T T::atan2(T & y)`
	$\sin(\pi x)$	`OP::sinpi`	`T T::sinpi()`
	$\cos(\pi x)$	`OP::cospi`	`T T::cospi()`
	$\arctan(x)/\pi$	`OP::atanpi`	`T T::atanpi()`
	$\text{atan2}(y, x)/\pi$	`OP::atan2pi`	`T T::atan2pi(T & y)`
Inverse	$\arcsin(x)$	`OP::asin`	`T T::asin()`
trigonometric	$\arccos(x)$	`OP::acos`	`T T::acos()`
functions	$\arctan(x)$	`OP::atan`	`T T::atan()`
Hyperbolic	$\sinh(x)$	`OP::sin`	`T T::sinh()`
functions	$\cosh(x)$	`OP::cos`	`T T::cosh()`
	$\tanh(x)$	`OP::tan`	`T T::tanh()`
Inverse	$\text{arsinh}(x)$	`OP::asin`	`T T::asinh()`
hyperbolic	$\text{arcosh}(x)$	`OP::acos`	`T T::acosh()`
functions	$\text{artanh}(x)$	`OP::atan`	`T T::atanh()`
Exponentials	e^x	`OP::exp`	`T T::expx()`
	$e^x - 1$	`OP::expm1`	`T T::expm1()`

Continued on next page...

Name	Operation	OP name	Signature
	2^x	`OP::exp2`	`T T::exp2()`
	$2^x - 1$	`OP::exp2m1`	`T T::exp2m1()`
	10^x	`OP::exp2`	`T T::exp10()`
	$10^x - 1$	`OP::exp10m1`	`T T::exp10m1()`
Logarithms	$\ln(x)$	`OP::log`	`T T::log()`
	$\log_2(x)$	`OP::log2`	`T T::log2()`
	$\log_{10}(x)$	`OP::log10`	`T T::log10()`
	$\ln(1 + x)$	`OP::logp1`	`T T::logp1()`
	$\log_2(1 + x)$	`OP::log2p1`	`T T::log2p1()`
	$\log_{10}(1 + x)$	`OP::log10p1`	`T T::log10p1()`
hypot	$\sqrt{x^2 + y^2}$	`OP::hypot`	`T T::hypot(T & y)`
Compound	$(1 + x)^n$	`OP::comp`	`T T::comp(T & y)`
Error	$\mathrm{erf}(x)$	`OP::erf`	`T T::erf()`
function	$\mathrm{erfc}(x)$	`OP::erfc`	`T T::erfc()`
Gamma	$\Gamma(x)$	`OP::gam`	`T T::gam()`
function	$\ln \lvert\Gamma(x)\rvert$	`OP::lgam`	`T T::lgam()`

Table B.1: *Operations known to `IeeeCC754++`.*

Table B.1 requires some further explanations: `T` denotes an appropriate subclass of `DriverFloatRepr`. x and y are floating-point values whereas n is an integer. "⇒" and "⇔" denote conversions between formats, usually between floating-point (fp) and integer (int) values or between decimal and binary representation.

Note that the first parameter x is always encoded into `*this`, i. e. inside the current instance of the appropriate subclass of `DriverFloatRepr`. Whenever the argument of the floating-point operator is an integer value (e. g. for the `pown` and `rootn` functions), this integer is encoded into a variable of type `T` by using the bit field that normally contains `IeeeCC754++`'s internal floating-point format representation. In other words, all operators always take one to three instances of `DriverFloatRepr` (or appropriate subclasses) as parameters, the first being an implicit parameter; the meaning of the number encoded in its bit field is context dependent.

Also note that `DriverFloatRepr` contains a pointer to a string only used to store the decimal representation for the conversions between a binary floating-point format and its decimal equivalent, namely `b2d` and `d2b`.

For the conversions between integer and floating-point values, the following naming conventions using two characters apply: The first character consists either of a "`c`" (**c**onvert from integer values) or an "`r`" (**r**ound to integer value). The second character denotes the integer format involved: "`u`" for **u**nsigned and "`i`" for signed **i**nteger values and capitalisation differentiating between 32 bit (small character) and 64 bit (capital character) formats.

The conversion functions `cX` (with `X` one of `i`, `u`, `I`, or `U`) take the specifications

of the target floating-point format as three integer parameters: the size `e` of the exponent, the size `m` of the significand (`m` for mantissa), and an integer `h` that denotes whether the target format uses a hidden bit (for $h > 0$).

Handling the registration process of operations and rounding modes is implemented in `FPregistry`. The following methods are available:

```
class FPregistry
{
  public:
    // handling rouding modes and operations simultaneously
    bool isRegistered(OP::op o, RD::rd r);
    bool clearall();

    // operations
    bool registerOP(OP::op);
    bool unregisterOP(OP::op);

    bool registerOPall();
    bool clearOPall();

    bool OPisRegistered(OP::op o) const;

    // rounding modes
    bool registerRD(RD::rd);
    bool unregisterRD(RD::rd);

    bool registerRDallIEEE();
    bool registerRDall();
    bool clearRDall();

    bool RDisRegistered(RD::rd r) const;
};
```

Operations can be registered by calling `registerOP()` and removed from the registry with `unregisterOP()`. It is also possible to add or remove all known operations at once with `registerOPall()` and `clearOPall()`. Furthermore, it can be checked via `OPisRegistered()` if the given operation is currently registered or not.

For the handling of rounding modes, an almost identical set of methods exists, with one notable addition: There are two methods to register "all" rounding modes. The distinction between `registerRDall()` and `registerRDallIEEE()` is mainly due to historic reasons: The latter registers all five rounding modes defined in IEEE 754-2008, while the former only registers the four rounding modes defined in IEEE 754 and IEEE 854, i. e. all rounding modes with the exception of roundTiesToAway.

Note that all methods taking an argument of either type `OP` or type `RD` exist in two or three variants, shown here for the respective register methods:

```
class FPregistry
{
    bool registerOP(OP::op);
    bool registerOP(char * s);
```

```
    bool registerRD(RD::rd);
    bool registerRD(char * s);
    bool registerRD(char r);
};
```

This makes it possible to register operations either with a parameter of type `OP` or with a string and rounding modes with a parameter of type `RD`, a string, or even a single character. The following code shows the equivalent ways of registering the addition operation and the rounding mode roundTowardZero:

```
FPreg.registerOP(OP::add);
FPreg.registerOP("add");

FPreg.registerRD(RD::zero);
FPreg.registerRD("zero");
FPreg.registerRD('z');
```

Table B.2 shows the possible values for the corresponding `RD` or character parameters to register rounding modes.

Rounding mode	`RD` name	character
roundTiesToEven	`RD::near`	`n`
roundTiesToAway	`RD::away`	`a`
roundTowardPositive	`RD::up`	`u`
roundTowardNegative	`RD::down`	`d`
roundTowardZero	`RD::zero`	`z`

Table B.2: *Rounding modes known to `IeeeCC754++`.*

Finally, there are two methods to work on operations and rounding modes at the same time: `clearall()` removes all entries from the registry, and `isRegistered()` checks whether the combination of operation and rounding mode contained in the current test vector is registered or not.

B.8.2 Enabling vector FPUs

To correctly handle FPUs working on SIMD vectors, `IeeeCC754++`'s vector capabilities must be explicitly enabled. This is achieved by calling `useVectorUnit()` inside `Register()`:

```
    useVectorUnit();
```

Two examples of `Register()` implementations, one of them setting up a vector FPU, can be found in the next section.

B.9 Example code for the new architecture and FPU

In this section, we summarise the process of implementing the `main` and an additional FPU by showing code for the `xyz` architecture and the `abc` FPU. The methods handling the floating-point environment are only implemented for the `main` FPU that supports the addition operation for all rounding modes (except roundTiesToAway, see above). The `abc` FPU only supports the square root operation in roundTiesToEven rounding mode. For brevity reasons, both FPUs only support operands in double precision. It is assumed that the `abc` FPU is a SIMD vector unit working on registers of 256 bit length which means every operand inside this FPU operates on four floating-point values at once.

The implementation of the corresponding files `src/xyz/fpu_main.cc` and `src/xyz/fpu_abc.cc` can be found in Listings B.2 and B.3.

```
// handle the floating-point environment - use C99 functions
void DriverFloat_main::SetLibRound()
{
    switch (GetFPRound())
    {
      case RM_NEAR: fesetround(FE_TONEAREST); break;
      case RM_ZERO: fesetround(FE_TOWARDZERO); break;
      case RM_UP:   fesetround(FE_UPWARD); break;
      case RM_DOWN: fesetround(FE_DOWNWARD); break;
    }
}

void DriverFloat_main::ClearLibExceptions()
{
    feclearexcept(FE_ALL_EXCEPT);
}

void DriverFloat_main::GetLibExceptions()
{
    if (fetestexcept(FE_DIVBYZERO)) SetFPDivByZero();
    if (fetestexcept(FE_INVALID)) SetFPInvalid();
    if (fetestexcept(FE_UNDERFLOW)) SetFPUnderflow();
    if (fetestexcept(FE_OVERFLOW)) SetFPOverflow();
    if (fetestexcept(FE_INEXACT)) SetFPInexact();

    ClearLibExceptions();
}

// register operations and rounding modes
void DriverFloat_main::Register()
{
    FPreg.clearall();
    FPreg.registerOP(OP::add);
    FPreg.registerRDall();
}

// example implementation for addition
DriverFloat_main DriverFloat_main::operator + (DriverFloat_main & m)
{
    if (isIEEEbinary64())
    {
        double dres, dop1, dop2;
```

```
        dop1 = todouble();
        dop2 = m.todouble();

        SetLibEnvironment();
        dres = dop1 + dop2;
        GetLibExceptions();

        DriverFloat_main rd(dres);
        return rd;
    }

    return skipTest();
}
```

Listing B.2: *Example implementation of* `DriverFloat_main` *for the* `xyz` *architecture.*

```
// register operations and rounding modes
void DriverFloat_abc::Register()
{
    FPreg.clearall();
    FPreg.registerOP(OP::sqrt);
    FPreg.registerRD(RD::near);

    useVectorUnit();
}

DriverFloat_abc DriverFloat_abc::sqrt()
{
    if (isIEEEbinary64())
    {
        double dop1;
        double dx1[4], dres[4];

        dop1 = todouble();
        fillVec(dx1, dop1, 4);

        SetLibEnvironment();
        SIMD_sqrt(dres, dx1);
        GetLibExceptions();

        if (checkVec(dres, 4))
        {
            setVectorError();
        }

        DriverFloat_abc rd(dres[0]);
        return rd;
    }

    return skipTest();
}
```

Listing B.3: *Example implementation of* `DriverFloat_abc` *for the* `xyz` *architecture.*

The vector implementation of `DriverFloat_abc::sqrt()` as shown in Listing B.3 requires some further explanation: `dop1` is used to convert the operand

(which is encoded in IeeeCC754++'s internal floating-point format) into a C++ **double** via todouble(). dx1 and dres denote vectors of four doubles length that are used for the operand and the result, respectively.

The call to fillVec() copies the floating-point value contained in dop1 into the vector dx1, whereas SIMD_sqrt() denotes the fictitious vector square root operation of the abc FPU.

After the execution of the vector operation, the resulting vector needs to be checked for consistency, i. e. checkVec() is called to verify that all four values contained in dres are identical. If this is not the case, a vector error is set by calling setVectorError(). Finally, the first entry in the result vector dres is passed to IeeeCC754++ for the regular conformity checks performed on every test vector.

B.10 Setting up main()

At this point, almost all code necessary to test the architecture xyz and the FPU abc has been added. As a last step, the main program which is contained in main() in src/xyz/main_xyz.cc must be given knowledge of the new FPU. We again modify the corresponding file copied from the dummy architecture. Code to call the main FPU is already present, as well as code performing general command line parameter handling.

The desired goal of adding the abc FPU can be achieved by textually replacing all occurrences of "GENERIC" and "generic" with "ABC" and "abc". The resulting code is shown and explained below:

```
#if defined BUILD_FPU_ABC || defined BUILD_FPU_ALL
  #include <DriverFloat_abc.h>
#endif
    ...
// main function for IeeeCC754 on default arch.
int main(int argc, char * argv[])
{
    ...
    std::string fpu = scanArg("fpu", argc, argv);
    if (!fpu.empty())
    {
        bool found = false;

#if defined BUILD_FPU_ABC || defined BUILD_FPU_ALL
        // this code will only be executed if the "abc" FPU is known
        // to the build system.
        if (fpu == "abc")
        {
            found = true;
            std::cout << "Calling code for \"abc\" fpu." << std::endl;

            return main_filehandler<DriverFloat_abc>(argc, argv);
        }
```

```
#endif
    }
    ...
}
```

Before the definition of `main()`, the header file for the `abc` FPU needs to be included when it was requested to be built either explicitly or by requesting all available FPUs (i. e. via one of the calls `configure -enable-fpu-abc` or `configure -enable-fpu-all`). Inside `main()`, the command line parameters need to be checked for an `--fpu` argument in order to determine the FPU being used during the testing process. Furthermore, additional setup code can be added here, as well as further parameter scanning possibly needed to initialise or choose an accelerator unit. Code to retrieve the command line parameters already exists in `src/xyz/main_xyz.cc`; the code which actually performs the call to the `abc` FPU is shown above.

B.11 Setting up and building the new architecture

After preparing the build system and adding code for the new architecture, the build system configuration files, such as `configure.ac` and `src/xyz/Makefile.am` must be compiled into a `configure` script and a `Makefile`. Additionally, some internal steps are performed such as adding appropriate preprocessor macros to `src/config.h.in`. This compilation step is performed by calling the bootstrap script in the source directory:

```
> ./bootstrap
```

At this point, `IeeeCC754++` can be compiled for the architecture `xyz` and the FPU `abc`, and the contained `main` and `abc` FPUs can be checked for IEEE-conformity. Listing B.4 sums up the configuration, build, and execution process for the new architecture.

```
> ./bootstrap
+ aclocal -I config
+ autoheader
+ automake --add-missing --copy
+ autoconf
> mkdir build
> cd build
> ../configure --enable-arch-xyz --enable-fpu-abc
configure: loading site script /usr/share/site/x86_64-unknown-linux-gnu
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking target system type... x86_64-unknown-linux-gnu
    ...
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
```

```
config.status: creating src/common/Makefile
    ...
config.status: creating src/xyz/Makefile
config.status: creating src/config.h
config.status: executing depfiles commands
configure:

    Build summary:
    --------------

    Build tests?   no
    Use hashing?   yes

    Compilers:     CC=gcc, CXX=g++ (g++)
    32/64 bit:     native
    Cross compile? no
    Default flags: no

    Modes:         main
    Architecture:  xyz
    FP units:      abc

    Compilation flags:
        CFLAGS:
        CPPFLAGS:
        CXXFLAGS:
        LDFLAGS:
        LIBS:      -lssl -lcrypto


> make
Making all in src
    ...
Making all in xyz
make[3]: Entering directory '/tmp/ttt/svn/build/src/xyz'
  CXX      fpu_main.o
  CXX      fpu_abc.o
  CXX      main_xyz.o
  CXXLD    IeeeCC754++_xyz
make[3]: Leaving directory '/tmp/ttt/svn/build/src/xyz'
> ./src/xyz/IeeeCC754++_xyz -vio ../testsets/alld -f xyz.log
Calling main implementation (no FPU kernel requested).
Using logfile: xyz.log
    ...
> ./src/xyz/IeeeCC754++_xyz -vio ../testsets/alld -f abc.log --fpu=abc
Calling code for "abc" fpu.
Using logfile: abc.log
    ...
>
```

Listing B.4: *Building* `IeeeCC754++` *for the architecture* `xyz` *and the FPU* `abc`.

Appendix C

Reference material

This appendix contains reference material for `IeeeCC754++`, the evaluation framework, and the optimisation framework, such as tables, usage information, example task files, and example implementations of analysis and fitness modules.

C.1 `fma` example

In this section, we give an alternative representation of the `fma` example discussed in Section 4.2.1 able to be used in distinguishing between fused and non-fused versions of a multiply-add operator. All numbers here are shown as binary representation of the numbers encoded in IEEE 754-2008 single precision format. Listing C.1 shows the test vector used in this example, including the three operands, the correct result in hexadecimal notation, the roundTiesToEven rounding mode and no exceptions being expected.

```
fmas n eq - 3f800001 3fc00000 bf800001 3f000001
```

Listing C.1: *`fma` test vector in UCB format, single precision.*

Multiplying the first two operands x and y with infinite intermediate precision produces the intermediate result i_1 (the hexadecimal operands from the UCB test vector are converted into binary representation, with lines denoting the borders between sign bit, exponent, significand, and trailing bits where applicable):

$$\begin{aligned} i_1 := x * y = \;& 0|01111111|00000000000000000000001 \\ & * 0|01111111|10000000000000000000000 \\ = \;& 0|01111111|10000000000000000000001|1 \end{aligned} \tag{C.1}$$

Still computing with infinite intermediate precision, adding z to i_1 yields the intermediate result i_2:

$$\begin{aligned} i_2 := i_1 + z = \quad & 0|01111111|10000000000000000000001|1 \\ & + 1|01111111|00000000000000000000001 \\ = \quad & 0|01111110|00000000000000000000001 \end{aligned} \tag{C.2}$$

The last bit in i_1 is cancelled out when adding z, and therefore i_2 is representable in single precision, yielding the final result $r_{\texttt{fma}} = i_2$.

On the other hand, when computing $r = x * y + z$ with regular floating-point operations (called `mla` here for multiply-add), i. e. when rounding after computing $x * y$, the situation changes (here, $RN : \mathbb{R} \to \mathbb{F}^*(\beta, t, L, U)$ denotes the roundTiesToEven rounding function): The last bit has to be rounded upwards, resulting in

$$i_3 = RN(i_1) = 0|01111111|10000000000000000000010. \tag{C.3}$$

The final addition of z then yields

$$\begin{aligned} i_4 := i_3 + z = \quad & 0|01111111|10000000000000000000010 \\ & + 1|01111111|00000000000000000000001 \\ = \quad & 0|01111110|00000000000000000000010 \end{aligned} \tag{C.4}$$

Again, i_4 is representable as a single precision number, so $r_{\texttt{mla}} = i_4$. Obviously, $r_{\texttt{fma}} \neq r_{\texttt{mla}}$.

C.2 `IeeeCC754++` usage

For quick reference, we list the syntax of the `IeeeCC754++` executable in Listing C.2. The calling conventions for the classic mode slightly differ from the other modes; it can be found in the next section.

```
> IeeeCC754++_default --help
Usage is: IeeeCC754++ <mode> <testset> [-f <logfile>] [ARGS ...]
  where:
          <mode>      Test mode.
          <testset>   Set of vectors which should be tested.
          <logfile>   Alternative log file..
To see an explanation of modes, call IeeeCC754++ --modes.
For information on common ARGS, call IeeeCC754++ --args.
> IeeeCC754++_default --modes
IeeeCC754++ modes:
  -c     Classic IeeeCC754; expects test vectors in Coonen format.
  -u     Classic IeeeCC754.
  -o     Classic IeeeCC754; converts test vectors from Coonen to UCB format.
  -s     Checksum mode.
  -d     Checksum mode, exceptions dropped.
  -h<mode>  Fingerprint mode, hashed.
       <mode> is hash digest; default is SHA1.
  -m     Fingerprint mode with hash digest MD5. Equivalent to -hmd5.
  -i     Grid info mode.
  -v<mode>  Verbose mode.
       <mode> is one of:
```

```
         io   Default. Print input and output vectors.
         in   Print only input vectors.
         out  Print only output vectors.
         ix   Print input and output vectors; count drop exception only errors.
         cc   Do not actually test, but print (expected) correct solution.
  -q     Quiet mode. Like "-u" but discarding output.

Where not otherwise specified, IeeeCC754++ expects test vectors in UCB format.

Please note that the calling syntax is different for the classic IeeeCC754 modes.
For more information, call IeeeCC754++ --classic.
> IeeeCC754++_default --args
IeeeCC754++ common options:
  --help             Displays basic usage.
  --modes            Lists all available modes.
  --args             This list of common options.
  --options          Identical to --args.
  --helpclassic      Displays basic syntax for classic modes.
  --classic          Identical to --helpclassic.
  --version          Show information on the current version.
  --ftz              Enable FTZ mode (flush to zero).
  --ftzsigned        Enable FTZ mode (flush to signed zero).
  --noftz            Disable FTZ mode.
  --skipsubnormal    Skip test vectors containing subnormals.
  --skiptiny         Identical to --skipsubnormal.
  --ulp=<ULP>        Set ULP threshold to <ULP>.
  --digest=<mode>    Generates an additional file <logfile>.<mode> containing
                     a (binary) hash digest of <logfile>. If <mode> is empty,
                     the default digest sha1 is used.
  --hexdigest=<mode> Generates an additional file <logfile>.<mode> containing
                     a (hexadecimal) hash digest of <logfile>. If <mode> is
                     empty, the default digest sha1 is used.
  --nolog            Suppresses generation of <logfile>.
```

Listing C.2: *IeeeCC754++ usage and command line parameters.*

C.3 IeeeCC754++ classic mode usage

Since `IeeeCC754++`'s classic mode is (almost) identical to `IeeeCC754`, we list the calling syntax of the latter as documented in `IeeeCC754`'s original `readme.usage` file for reference purposes:

```
How to call the driver program IeeeCC754
========================================

Summary:

IeeeCC754  -c {-s|-d|-l|-q|-m|{-e <int> -t <int> [-h]}}
              -r {n|p|m|z}  -i -n {i|o|x|z|u|tiny|nan|inf|snz} -x
              -f logfile  testfile

IeeeCC754  -u -r {n|p|m|z} -i -n {i|o|x|z|u|tiny|nan|inf|snz} -x
              -f logfile  testfile

IeeeCC754  -o <file> {-s|-d|-l|-q|-m|{-e <int> -t <int> [-h]}}
              -r {n|p|m|z}  -i -n {i|o|x|z|u|tiny|nan|inf|snz}
              -f logfile  testfile

The required argument 'testfile' is a file of test vectors in extended
Coonen syntax (the default) or SUN-UCB format.

The options of the driver program can be subdivided in three categories,
listed below.

(1) Options to specify the purpose of the run. Use precisely one of the
    following three options:
         -c : perform testing: file of test vectors is in Coonen syntax
         -u : perform testing: file of test vectors is in SUN-UCB format
         -o <file> : do not perform testing but only translate testdata from
                     Coonen syntax to hexadecimal SUN-UCB format and output
                     to <file>
```

```
(2) Options to specify the precision and exponent range of the source (and
    for some conversions also the destination) floating-point format, when
    the file of test vectors is in extended Coonen syntax.
    When the file of test vectors is in SUN-UCB format, the precision and
    exponent range are specified in the testfile itself, and the options
    below should be skipped.
         -s : single precision (same as -e 8 -t 24 -h)
         -d : double precision (same as -e 11 -t 53 -h)
         -l : long double precision (same as -e 15 -t 64)
         -q : quadruple precision (same as -e 15 -t 113 -h)
         -m : 240 bit multiprecision (same as  -e 15 -t 240)
         -e <int> : provide <int> bits to represent exponent
         -t <int> : provide <int> bits precision
         -h : leading bit is hidden
         -ds : single precision destination
         -dd : double precision destination
         -dl : long double precision destination
         -dq : quadruple precision destination
         -dm : 240 bit multiprecision destination
         -de <int> : size of destination exponent
         -dt <int> : destination precision
         -dh : leading bit is hidden in destination

(3) Options to influence the actual testing phase (optional)
         -r {n|p|m|z}: test only the specified rounding modes
         -n {i|o|x|z|u|tiny|nan|inf|snz}: do not test the specified
                  exceptions, denormalized numbers, NaNs, signed
                  infinities or signed zeroes
         -j {o|u|i|z}: jump/skip test vectors raising the overflow, underflow,
                  invalid or divide by zero exception
         -ieee: test conversions only within range specified by IEEE
         -i: idem as -ieee
         -x: use extended precision on x86 instead of a (forced) single
                  or double precision
         -f logfile: output log of testing to 'logfile'; the default value
                  for 'logfile' is ieee.log
```

Listing C.3: *readme.usage*

C.4 Error codes used in `IeeeCC754++`

Tables C.1 and C.2 show the error codes used in `IeeeCC754++`'s verbose mode (cf. Section 3.3.2) and their explanation. For quick lookup, Table C.1 is sorted alphabetically, whereas Table C.2 groups error codes according to their meaning.

shortname	error description
a	fma error
b	underflow before rounding previously not detected
c	underflow before rounding previously detected
d	different decimal representation
e	exponent different
f	flush to zero detected
g	result is NaN but expected infinity
h	result is not an infinity
i	invalid not expected
j	inexact flag not returned
k	overflow flag not returned

Continued on next page...

shortname	error description
l	underflow not returned
m	mantissa (significand) different
n	result is not a NaN
o	overflow not expected
p	invalid flag not returned
q	divide flag not returned
r	scalar error (not all no-scalar fields left alone)
s	Different sign
t	result is normalised number but expected subnormal
u	underflow without denormalisation loss previously not detected
v	vector error (not all returned floats equal)
w	underflow without denormalisation loss previously detected
x	inexact not expected
y	underflow not expected
z	divide by zero not expected
>	vector exception error (not all returned exceptions equal)

Table C.1: *Error short codes used in IeeeCC754++'s verbose output format and their respective description.*

shortname	error description
z	divide by zero not expected
i	invalid not expected
x	inexact not expected
o	overflow not expected
u	underflow without denormalisation loss previously not detected
b	underflow before rounding previously not detected
y	underflow not expected
j	inexact flag not returned
k	overflow flag not returned
l	underflow not returned
w	underflow without denormalisation loss previously detected
c	underflow before rounding previously detected
p	invalid flag not returned
q	divide flag not returned
d	different decimal representation
n	result is not a NaN
s	Different sign
e	exponent different

Continued on next page...

shortname	error description
m	mantissa (significand) different
v	vector error (not all returned floats equal)
>	vector exception error (not all returned exceptions equal)
r	scalar error (not all no-scalar fields left alone)
f	flush to zero detected
t	result is normalised number but expected subnormal
g	result is NaN but expected infinity
h	result is not an infinity
a	fma error

Table C.2: *Error short codes used in `IeeeCC754++`'s verbose output format and their respective description.*

C.5 Reference task files

In this section, we show reference task files for use with the evaluation framework (Listings C.4, C.5, C.6, and C.7 and the optimisation framework (Listing C.8). For details on the evaluation framework and the optimisation framework, as well as how to use the task files within the frameworks, see Sections 3.4 and 3.5.

```
# Reference compilation file
#
# Format is:
#  - all lines beginning with "#" are comments
#  - comments and empty lines (i.e. lines only containing whitespace) are ignored
#  - lines must have the form "KEY VAL"
#  - KEY is on of the following characters:
#      j  <FILE>    job
#      c  <FILE>    compile
#      t  <FILE>    test
#      e  <FILE>    eval
#      m  v|x|c     mode
#      d  <NAME>    delimiter
#  - VAL must be a filename (shown as <FILE> above>, with or without the
#    proper ending.
#  - The correct ending (".job for job, ".com" for compile, ".test" for test,
#    ".eval" for eval tasks) will be appended if not already specified.
#  - Tests will be executed in a subdir MYNAME.DATE where MYNAME is the name
#    of this job file without ".job".
#  - For KEY = m, VAL must be one of v (verbose mode), x (verbose mode ignoring
#    exceptions), or c (classic mode).
#  - For KEY = d, a delimiter is printed into the log file, including the given
#    string <NAME> as heading.
#
# The job script will read each line. For "j", "c", "t" and "e" lines, it
# reads the file VAL.<END> and starts the corresponding tasks (job, compile,
# test, or eval task).
# For "m" lines, the testing mode is changed, and for "d" lines, a delimiter
# is printed into the output log file (see above).

# This reference file simply takes reference tasks and executes them one after
# the other. The corresponding files are called ref.com, ref.test, and
# ref.eval.

# show delimiter including the heading "REFERENCE - EVALUATION FRAMEWORK"
d REFERENCE - EVALUATION FRAMEWORK

# compile IeeeCC754++
c ref
```

```
# change IeeeCC754++ testing mode to verbose
m v

# execute test and eval tasks
t ref
e ref

# optional - use opt job to run optimization framework
#
# WARNING: Only one opt task is allowed per evaluation framework run.
# If more than one optimisation job is started, problems will arise!

# show delimiter including the heading "REFERENCE - OPTIMIZATION FRAMWORK"
d REFERENCE - OPTIMIZATION FRAMWORK

# the opt task
o ref
```

Listing C.4: *files/ref.job*

```
# Reference compilation file
#
# Format is:
#  - all lines beginning with "#" are comments
#  - comments and empty lines (i.e. lines only containing whitespace) are ignored
#  - lines must have the form "KEY = VAL"
#  - lines will be split at first "="
#  - KEY and VAL will be trimmed (whitespace at beginning/end removed)
#  - unknown KEYs will be ignored, but a warning will be printed
#  - case of KEY is irrelevant
#  - case of VAL is taken "as is"
#
# The evaluation framework will take these values and execute the following
# commands:
#
# $> source <ENV>
# $> configure --prefix=<PATH> CC=$MYCC CXX=$MYCXX \
#      CFLAGS=<CFLAGS> CPPFLAGS=<CPPFLAGS> CXXFLAGS=<CXXFLAGS> LDFLAGS=<LDFLAGS> LIBS=<LIBS> \
#      --enable-arch=<ARCH> --enable-mode-<MODE>(s) --enable-fpu-<FPU>(s) \
#      --enable-m<BITS> <ARGS>
# $> make -j<CORES>
# $> make install

# Setup the environment for the compile task. This is done by sourcing a shell
# script that exports all needed env variables.
#
# Important: This script should export MYCC and MYCXX variables as these will
# be used as the compiler names!
#
# If more than one environment script is needed, supply as ENV entries on
# separate lines:
# ENV = script1
# ENV = script2
#
# Mandatory: no
# Default:   none

ENV =

# Alternatively setup the environment for the compile task via the modules
# system. This is done by supplying a name of the module that will be loaded
# via "module load <module>".
#
# Important: This script should export MYCC and MYCXX variables as these will
# be used as the compiler names!
#
# If more than one module is needed, supply as MODULE entries on
# separate lines:
# MODULE = module1
# MODULE = module2
#
# Mandatory: no
# Default:   none

MODULE =

# Name of the platform's architecture (as known by IeeeCC754++).
#
```

```
# Mandatory: yes
# Default:   none

ARCH = default

# List all FPUs that need to be built. "all" builds all FPUs.
#
# Mandatory: no
# Default:   none

FPU =

# In the following, extra environment flags can be given if needed.
#
# Mandatory: no
# Default:   none

CFLAGS =
CPPFLAGS =
CXXFLAGS =
LDFLAGS =
LIBS =

# List extra arguments for the configure script here.
#
# Mandatory: no
# Default:   none

ARGS =

# Number of cores that will be used during the build stage.
#
# Mandatory: no
# Default:   1

CORES =

# List all modes to be built.
#
# Mandatory: no
# Default:   main

MODE =

# BITS specifies if a 32 or 64 version should be built. If no value is given,
# a native build is performed.
#
# Mandatory: no
# Default: none

BITS =
```

Listing C.5: *files/ref.com*

```
# Reference test file
#
# Format is:
#  - all lines beginning with "#" are comments
#  - comments and empty lines (i.e. lines only containing whitespace) are ignored
#  - lines must have the form "KEY = VAL"
#  - lines will be split at first "="
#  - KEY and VAL will be trimmed (ws at beginning/end removed)
#  - unknown KEYs will be ignored, but a warning will be printed
#  - case of KEY is irrelevant
#  - case of VAL is taken "as is"
#
# The evaluation framework will take these values and execute the
# specified test task basically as follows (for every TESTSET and FPU):
#
# $> source <ENV>
# $> IeeeCC754_<ARCH> -v <SRC>/<TESTSET> -f MYNAME_DATE.log --fpu=<FPU> <ARGS>

# Setup the environment for the test task. This is done by sourcing one or
# more shell scripts that export all needed env variables.
#
# If more than one environment script is needed, supply as ENV entries on
# separate lines:
# ENV = script1
```

```
# ENV = script2
#
# Important: Most likely, the same setup as for the compile task is needed
# in order to use the correct shared libraries that are linked into the
# executable. However, the environment scripts used in the COMPILE script
# (see below) are not imported from the COMPILE file, so they must be
# explicitely specified here.
#
# Mandatory: no
# Default:   none

ENV =

# Alternatively setup the environment for the test task via the modules
# system. This is done by supplying a name of the module that will be loaded
# via "module load <module>".
#
# Important: This script should export MYCC and MYCXX variables as these will
# be used as the compiler names!
#
# If more than one module is needed, supply as MODULE entries on
# separate lines:
# MODULE = module1
# MODULE = module2
#
# Mandatory: no
# Default:   none

MODULE =

# List the name of the compile task here (which is specified in the file
# TASK.com). All necessary values (which are not overwritten here) will be
# retrieved from that task file.
#
# Mandatory: yes
# Default:   none

COMPILE =

# The names of the testsets that contain the test vectors.
#
# These testsets need to be located in <SRC>/testsets/ (which is the standard
# path for testsets inside IeeeCC754++'s source tree).
#
# Mandatory: yes
# Default: none

TESTSET =

# If we want to test an FPU, specify it here. If not specified, the main FPU
# implementation will be used.
#
# PLEASE NOTE: In contrast to compile task files, FPU is not a list, so only
# a single FPU entry is allowed here!
#
# Mandatory: no
# Default:   main

FPU =

# List extra arguments for the test executable here.
#
# Mandatory: no
# Default:   none

ARGS =

# For execution via a batch system, specify the batch system module here.
#
# The following setup (which is typical for HPC systems) is assumed:
# - Compilation is done on a front end node (possibly using a cross compiler).
# - The tests are executed on compute nodes and submitted via a batch system.
#
# This results in the follwing execution setup:
# - The evaluation framework is started on the frontend.
# - All tasks except the test task are performed on the frontend.
# - Tests are executed via the following process:
#   * submit test to batch system
#   * poll batch system until job is finished
#   * retrieve results (possibly done implicitely)
#
# Known batch systems:
```

```
#   ll     (LoadLeveler)
#   pbs    (Torque/PBS)
#   slurm  (SLURM)
#
# Mandatory: no
# Default:   none

BATCH =

# If IeeeCC754++ cannot be executed directly (e.g. because it must be started
# via mpiexec or similar), EXECPREFIX can be used to specify the necessary
# command with with the IeeeCC754++ execution line will be prefixed.
#
# For example, "EXECPREFIX = mpiexec -n 2" results in an execution call to
# "mpiexec -n 2 IeeeCC754++ <OPTIONS...>".
#
# Mandatory: no
# Default:   none

EXECPREFIX =
```

Listing C.6: *files/ref.test*

```
# Reference test file
#
# Format is:
#  - all lines beginning with "#" are comments
#  - comments and empty lines (i.e. lines only containing whitespace) are ignored
#  - lines must have the form "KEY = VAL"
#  - lines will be split at first "="
#  - KEY and VAL will be trimmed (ws at beginning/end removed)
#  - unknown KEYs will be ignored, but a warning will be printed
#  - case of KEY is irrelevant
#  - case of VAL is taken "as is"
#
# The evaluation framework will take these values and execute the
# specified evaluation task by applying each evaluation function to
# the test results.

# List of evaluation functions to be used to summarise results.
# A list of valid evaluation functions can be retrieved with the following
# commands:
# > python job.py --list
# > python eval.py --list
#
# Mandatory: no
# Default:   basic

EVALFUNCTION =

# List the name of the test task here (which is specified in the file
# TASK.test). All necessary values (which are not overwritten here) will be
# retrieved from that task file.
#
# Mandatory: no
# Default:   none

TEST = ref

# When TEST is not given, it is possible to supply a list of logfiles
# via LOGFILES and LOGPATH. These values are ignored when TEST is specified.
#
# Mandatory: no
# Default:   none

LOGPATH =
LOGFILES =
```

Listing C.7: *files/ref.eval*

```
# Reference optimization file
#
# Format is:
#  - all lines beginning with "#" are comments
#  - comments and empty lines (i.e. lines only containing whitespace) are ignored
#  - lines must have the form "KEY = VAL"
#  - lines will be split at first "="
#  - KEY and VAL will be trimmed (ws at beginning/end removed)
#  - unknown KEYs will be ignored, but a warning will be printed
#  - case of KEY is irrelevant
#  - case of VAL is taken "as is"
#
# The optimisation framework will take the values specified in this file
# and execute an optimisation run as follows:
#  - A list of compiler option combinations is generated.
#  - For every combination of compiler, fpu, and options, a number of tasks
#    is generated:
#     * a compile task
#     * a test task
#     * an eval task (with opteval eval function, see below)
#     * a timing task, using the specified external application
#  - The tasks are executed.
#  - For every fitness function that is specified, this function is
#    applied to all test results, and a result table is generated.
#  - Finally, the result tables are printed.

# Name of the platform's architecture (as known by IeeeCC754++).
#
# Mandatory: yes
# Default:   none

ARCH = default

# Name of the platform's architecture as displayed in results.
#
# Mandatory: no
# Default:   ARCH

ARCHNAME =

# List of compilers to be used.
#
# Mandatory: yes
# Default:   none

COMPILERS =

# If the environment needs to be set up via the modules system,
# supply the template for the module to be loaded.
#
# This assumes that the name of the compiler is given in the form
# [cn]-[cv] where [cn] is the compiler name and [cv] the compiler
# version. Alternatively, [c] can be used for the full compiler name.
#
# Example:
#   COMPILERS = gcc-7.1 my-compiler-2
#   MODULE = compiler/[cn]/[cv]
# results in the following module invocations:
#   module load compiler/gcc/7.1
#   module load compiler/my-compiler/2
#
# Mandatory: yes
# Default:   none

MODULE =

# List all FPUs that are to be tested.
#
# Mandatory: no
# Default:   main

FPU =

# The name of the testset that contains the test vectors.
#
# This testset needs to be located in <SRC>/testsets/ (which is the standard
# path for testsets inside IeeeCC754++'s source tree).
#
# Only one TESTSET entry is allowed for the optimisation framework.
#
# Mandatory: yes
```

```
# Default: none

TESTSET =

# Compiler flags that should be optimised. Basically, a hierarchical list
# of all combinations is generated as follows:
#   The flags
#     LEVEL1 = -a
#     LEVEL2 = -b1 -b2
#     LEVEL3 = -c1 -c2
#   results in these 8 combinations:
#     [no options]
#     -a
#     -a -b1
#     -a -b1 -c1
#     -a -b1 -c2
#     -a -b2
#     -a -b2 -c1
#     -a -b2 -c2
#
# NOTE 1: Not all levels need to be defined, but if one level stays empty,
# the following levels are ignored. This means that if LEVEL1 is empty,
# only one combination is tested, i.E. the one with empty options.
#
# NOTE 2: It is possible to test all combinations on levels 2 and 3
# => see COMBINATIONSLEVEL below.
#
# Mandatory: no
# Default:   none

LEVEL1 =
LEVEL2 =
LEVEL3 =

# When generating the list of compiler option combinations that should be
# tested, it is possible to generate all combinations of options on level
# 2 and 3. The following values for COMBINATIONSLEVEL are possible:
#
#   0 => no combinations used (default behaviour)
#   1 => combinations on last level (2 or 3)
#   2 => combinations on from level 2 on, i.e. on levels 2 and 3
#   3 => combinations only on level 3
#
# Mandatory: no
# Default:   0

COMBINATIONSLEVEL =

# In the following, extra environment flags can be given if needed.
#
# Mandatory: no
# Default:   none

CFLAGS =
CPPFLAGS =
CXXFLAGS =
LDFLAGS =
LIBS =

# The following APP_ variables are needed to evaluate the effect of the
# applied compiler options with respect to performance. This is done as
# follows:
# * To evaluate the performance/conformity tradeoff, performance levels
#   need to be retrieved.
# * As performance is highly dependent on the actual application, use
#   this application to measure real-world performance.
# * The script APP_BUILD (which should accept compiler and options via
#   environment flags) is used to build the external application.
# * The script APP_EXEC is executed to retrieve runtime of the application.
# * Ideally, runtime should be in the order of O(10) seconds as this limits
#   the overall runtime of the optimisation framework to reasonable levels
#   while still providing sufficiently long running time.
# * If more than one run is needed to get stable performance results,
#   APP_REPEATS can be used to specify the number if runs (at most 10).
# * The actual runtime can be measured either by the optimisation framework
#   internally (i.E. the runtime of APP_EXEC is measured) or by the
#   application itself (i.E. externally). In the latter case, the runtime
#   in seconds must be returned on the command line as the only (float)
#   value on the last line of output.
# * By default, HPCG 3.0 is used as the external application.
# * Additionally, "sixloops" can be used.
#
```

```
# Mandatory: no
# Defaults:  APP_BUILD   = hpcg_build_serial.sh
#            APP_EXEC    = hpcg_execute.sh
#            APP_REPEATS = 1
#            APP_TIMING  = internal
#
#
# example settings for the provided applications:
#
# HPCG parallel build (with MPI and OpenMP):
#
#     APP_BUILD   = hpcg_build_parallel.sh
#     APP_EXEC    = hpcg_execute_mpi.sh
#     APP_REPEATS = 3
#     APP_TIMING  = external
#
# sixloops built with environment modules:
#
#     APP_BUILD   = sixloops_mod_build.sh
#     APP_EXEC    = sixloops_mod_execute.sh
#     APP_REPEATS = 1
#     APP_TIMING  = external

# If the automatic retrieval of timing information during an
# optimisation framework run is not desired (e.g. because execution of a
# significand number of test runs is not feasible or because run times
# will be evaluated outside the optimisation framework), the variable
# USE_EXTERNAL_APP can be used to completely bypass the timing step,
# thereby speeding up the execution of the optimisation framework
# significantly.
#
# Mandatory: no
# Default:   yes

USE_EXTERNAL_APP =

# List of fitness functions to be used to rank results.
# A list of valid fitness functions can be retrieved with the following
# command:
# > python opt.py --list
#
# Mandatory: no
# Default:   success_rate

FITNESS =

# The name of the evaluation function that is to be used to summarize results.
#
# NOTE that for the optimisation framework to work as expected, the default
# choice "opteval" must be used as this specially tailored evaluation function
# retrieves summary values from the test results and feeds them into the
# optimisation db for further analysis with fitness functions (see above).
#
# In other words, it should not be necessary to change this value except when
# a custom evaluation function has been added for special purposes.
#
# Mandatory: no
# Default:   opteval

EVALFUNCTION =

# List extra arguments for the configure script here.
#
# Mandatory: no
# Default:   none

ARGS =

# Number of cores that will be used during the build stage.
#
# Mandatory: no
# Default:   1

CORES =

# List all modes to be built.
#
# Mandatory: no
# Default:   main

MODE =
```

```
# BITS specifies if a 32 or 64 version should be built. If no value is given,
# a native build is performed.
#
# Mandatory: no
# Default: none

BITS =
```

Listing C.8: *files/ref.opt*

C.6 Evaluation function example

Section 3.4.2 describes the concept of analysis modules used in the evaluation framework to enable quick and easy analysis of the logfiles generated by `IeeeCC754++` test runs, as well as the evaluation functions implemented in the evaluation framework. Since this selection of evaluation functions can never be exhaustive, the evaluation framework may be extended with custom analysis modules. For a description of this process, see Section 3.4.2, page 118. Listing C.9 shows an example implementation of a custom evaluation function. Note that this implementation is not functional in that it does not actually analyse or evaluate log files. Rather, it demonstrates common techniques able to be helpful when implementing custom analysis modules and describes in detail the contents of the test summary database.

```
#!/usr/bin/env python
#
# Author:   Matthias Huesken
# Purpose:
#
# $Id: example.py 872 2017-06-27 16:59:04Z huesken $
#

# importPySQLite needs to be imported to ensure SQLite works properly
import importPySQLite

# evaltools contains some useful utility functions:
# - printSep(<STRING>), printHugeSep(<STRING>):
#     return formatted seperators
# - getInt(<RESULT>):
#     return an integer as result of "select count(*)" operation on
#     the SQLite DB
import evaltools

# Every analysis module needs a version() method
def version():
    return "Example evaluation function v0.01"


# The evaluation function in an analysis module is implemented inside
# the method evaluate() which takes a SQLite table as the only paramater.
def evaluate(db):
    # The database that contains the result of the test runs contains of
    # exactly one table called "results" which is created with the
    # following command (shown here for reference purposes):
    #
    # db.execute("CREATE TABLE results(line not null, precision, operation, roundmode,
    # operand1, operand2, operand3, resultexp, resultret, exceptexp, exceptret,
    # success, errors, ulps, primary key(line))")
    #
    # with the following meaning for the entries:
    #
    #     line        Line number.
    #     precision   Floating-point format (h, s, d, q, l).
    #     operation   Operation that was executed.
```

```
#     roundmode   Rounding mode that was used for the operation.
#     operand1    The first operand.
#     operand2    The second operand (if there was one).
#     operand3    The third operand (currently only for fma).
#     resultexp   Expected result (i.e. the correct result of the
#                 operation as stored in the test vector).
#     resultret   Returned result (i.e. the result of the operation
#                 as returned by the FPU).
#     exceptexp   Expected exceptions (i.e. the correct exceptions as
#                 stored in the test vector - may be empty if no
#                 exceptions should be raised).
#     exceptret   Exceptions that were actually returned.
#     success     "1" if the returned result is identical to the
#                 expected result and if one of the following holds:
#                 - the returned exceptions are identical to the
#                   expected ones
#                 OR
#                 - settings.OUTPUTMODE is set to "x" (i.e. exceptions
#                 are ignored when evaluating the "success");
#                 "0" else.
#     errors      "1 - success", i.e. "1" if success equals "0" and
#                 vice versa.
#     ulps        Difference between expected and returned result in
#                 ULPs.
#
# To analyse the testing results, arbitrary SQL operations may be
# performed on db.results, followed by further analysis/evaluation/
# computation in Python.
#
# In the following, we give a few examples of common SQL queries and
# useful evaluation operations in Python. For more examples, see the
# source code of the other analysis modules.
#

# First, always get a pointer to the db.
c = db.cursor()

# Retrieve all operations that are contained in the db and print them
# to the console. Note that the returned values are tuples which
# contain the desired data in the first entry.
c.execute("SELECT DISTINCT operation FROM results ORDER BY line")
operations = c.fetchall()
for operation in operations:
    print "Found operation:", operation[0]

# Get overall and error counts and calculate error rate.
# Prints the rate the the console like this: " 23.45%"
overall = evaltools.getInt(c.execute("SELECT count(*) FROM results"))
error   = evaltools.getInt(c.execute("SELECT count(*) FROM results WHERE success = 0"))
print "Error rate: %6.2f%%" % (float(error) / float(overall) * 100.0)

# Same as before, with two differences: Calculate success rate
# only for all operations with "round to nearest", i.e. for all
# operations that were executed with roundTiesToEven mode.
# By setting "u", "d", "z". or "a" for "roundmode", results for
# the other rounding modes may be retrieved.
roundmode = "n"
overall = evaltools.getInt(c.execute("SELECT count(*) FROM results WHERE roundmode = ?",
     (roundmode,)))
success = evaltools.getInt(c.execute("SELECT count(*) FROM results WHERE success = 1 AND
     roundmode = ?", (roundmode,)))
print "Error rate: %6.2f%%" % (float(error) / float(overall) * 100.0)

# Finally, we show how the values needed by the analysis module
# "error_report" are retrieved:
#
# This query will return tuples with the following entries:
#
# (<ERROR TYPE>, <OPERATION>, <ULPS>, <COUNT>)
#
# where <COUNT> is the count of occurrences for the combination of
# error type and operation.
#
# Note that the "error_report" module further processes the
# returned tuples to generate the report. Converting the error
# types into readable text is done via the method getErrors(err)
# in the "error" module. For details, see the implementation in
# error_report.py.
#
c.execute("SELECT errors, operation, ulps, count(errors) FROM results WHERE success = 0 AND
     errors <> '' GROUP BY operation, errors ORDER BY errors, operation")
results = c.fetchall()
```

```
    # After processing, the report must be returned as a string.
    output = evaltools.printSep("Example evaluation function report")
    output += """Generated by: %s

Example evaluation function output - since this module does not
actually compute anything, simply return this text.

In a "real" evaluation function, the generated report should be returned
here as a string, i.e. formatted for easy reading.
""" % version()

    return output
```

Listing C.9: *evalfunc/example.py*

C.7 Fitness function example

Listing C.10 shows a small example that demonstrates how a custom fitness function (which are described in Section 3.5.2) can be added to the optimisation framework. It represents a working example of a fitness function, although it is not particularly useful: It simply retrieves the results from the result database and applies a fitness value to each value according to the order in which the results were retrieved from the database. However, the heavily commented file gives a description of the contents of the result database which can be useful when implementing custom fitness functions. The general process of extending the optimisation framework with a new fitness module is discussed in Section 3.5.2, page 142.

```
#!/usr/bin/env python
#
# Author:   Matthias Huesken
# Purpose:
#
# $Id: example.py 873 2017-06-27 17:00:24Z huesken $
#

# importPySQLite needs to be imported to ensure SQLite works properly
import importPySQLite

# fitnesstools contains some useful utility functions:
# - getInt(<RESULT>):
#     return an integer as result of a "select count(*)" operation on
#     the SQLite DB
# - getVectors(<DB>, <SQLSTATEMENT>):
#     fetch all vectors that are returned by executing <SQLSTATEMENT>.
#     If SQLSTATEMENT is empty, all lines are retrieved (SELECT * from optimiser)
# - setFitness(<DB>, <KEY>, <FITNESS>):
#     push fitness value <FITNESS> for vector with key <KEY> into <DB>
# - setFitness(<DB>, <FITNESSVECTOR>):
#     push fitness values (all contained in <FITNESSVECTOR>) into <DB>
import fitnesstools

# Every analysis module needs a version() method
def version():
    return "Example fitness function v0.01"


# The fitness function in a fitness module is implemented inside
# the method fitness() which takes an SQLite table as the only paramater.
def fitness(db):
    # The database that contains the result of the test runs consists of
    # exactly one table called "optimiser" which is created with the
    # following command (shown here for reference purposes):
    #
```

```
# db.execute("CREATE TABLE optimiser(name NOT NULL, options, numOptions INT,
# overall INT, success INT, errors INT, successRate FLOAT, errorRate FLOAT,
# successExcept INT, successExceptRate FLOAT, runtime FLOAT, fitness FLOAT,
# PRIMARY KEY(name))")
#
# with the following meaning for the entries:
#
#    name               Name of the currently tested set.
#    options            Compiler options used in the current set.
#    numOptions         Number of compiler options used.
#    overall            Number of test vectors.
#    success            Number of successfully executed test vectors.
#    errors             Number of test vectors that returned errors.
#    successRate        Succes rate.
#    errorRate          Error rate.
#    successExcept      Number of successes, ignoring exceptions.
#    successExceptRate  Success rate, ignoring exceptions.
#    runtime            Runtime of external application.
#    fitness            Fitness (to be determined!).
#
# The goal of a fitness function is to apply a fitness value to every
# entry (row) in the optimiser db. How this fitness is computed is up
# to the fitness function. Note that at the end of the fitness function
# (which should not return anything) these fitness values must be pushed
# into the db via setFitness() (see above).
#
# To analyse the results, arbitrary SQL operations may be
# performed on db.optimiser, followed by further analysis/evaluation/
# computation in Python. However, in most cases, it should be sufficient
# to retrieve the contents of the optimiser table via getVectors(),
# possibly sorted via a specifically crafted SQL statement, compute
# some fitness value, and push this back into the db.
#
# In the following, we give a few examples of common SQL queries and
# useful evaluation operations in Python. For more examples, see the
# source code of the other fitness modules.
#
# First, always retrieve the contents of the db.
# This call is equivalent to getVectors(db, "SELECT * FROM optimiser;")
rows = fitnesstools.getVectors(db)

# Or retrieve the contents sorted by some criteria.
# This example sorts the table according to "success" first and
# "runtime" second.
# Note that only the name is selected here since with this parameter,
# a fitness value can be written back into the db.
rows = fitnesstools.getVectors(db, "SELECT name FROM optimiser ORDER BY successRate DESC,
     runtime ASC, numOptions ASC;")

# For the above example, a unique fitness value appropriate for later
# sorting by the optimisation framework can be assigned as follows:
# * loop over the rows and assign a decreasing fitness value to the
#   current row
# * additionally, push this value into the db via setFitness()
fitness = 999.99
for row in rows:
    fitnesstools.setFitness(fitness)
    i -= 0.01

# For more examples, see the other fitness modules.
# For a much more elaborate example that actually computes a fitness
# value (in contrast to assigning one, see above), see the "weighted"
# fitness function implemented in weighted.py.
```

Listing C.10: *fitnessfunc/example.py*

List of Figures

List of Tables

List of Listings

Bibliography

[All15a] Allwinner Technology. *A20 Datasheet, Revision 1.5*. 06/04/2015. URL: https://github.com/allwinner-zh/documents/raw/master/A20/A20_Datasheet_v1.5_20150510.pdf (visited on 15/11/2017).

[All15b] Allwinner Technology. *A64 Datasheet, Revision 1.1*. 26/06/2015. URL: http://files.pine64.org/doc/datasheet/pine64/A64_Datasheet_V1.1.pdf (visited on 15/11/2017).

[AMD00a] AMD. *3DNow! Technology Manual*. 2000. URL: http://support.amd.com/TechDocs/21928.pdf.

[AMD00b] AMD. *AMD Extensions to the 3DNow! and MMX Instruction Sets – Manual*. 2000. URL: http://support.amd.com/TechDocs/21928.pdf.

[AMD05] AMD. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*. 2005.

[AMD17a] AMD. *AMD EPYC*. 2017. URL: http://www.amd.com/en/products/epyc (visited on 28/11/2017).

[AMD17b] AMD. *AMD Ryzen*. 2017. URL: http://www.amd.com/en/ryzen (visited on 28/11/2017).

[AMD17c] AMD. *Radeon Pro*. 2017. URL: https://pro.radeon.com/en/ (visited on 28/11/2017).

[AMD17d] AMD. *Server Graphics and Accelerators*. 2017. URL: http://www.amd.com/en/products/servers-graphics (visited on 28/11/2017).

[APMa] Applied Micro. *X-Gene X-C1 Evaluation Kit*. URL: https://www.apm.com/docs/X-C1_PB.pdf (visited on 17/11/2017).

[APMb] Applied Micro. *X-Gene X-C2 Evaluation Kit*. URL: https://myapm.apm.com/technical_documents/download/apm883408-x2-x-gene-2-evaluation-kit-product-brief (visited on 17/11/2017).

[APM17] Applied Micro. *X-Gene*. 2017. URL: https://www.apm.com/products/data-center/x-gene-family/x-gene/ (visited on 15/11/2017).

[ARMa] ARM. *Arm Instruction Emulator*. URL: https://developer.arm.com/products/software-development-tools/hpc/arm-instruction-emulator.

[ARMb] ARM. *ARM Performance Libraries*. URL: https://developer.arm.com/products/software-development-tools/hpc/arm-performance-libraries.

[ARM05] ARM. *ARM Architecture Reference Manual*. 2005. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0100i/index.html.

[ARM14] ARM. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. 2014. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html.

[ARM17] ARM. *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. 2017. URL: https://developer.arm.com/docs/ddi0487/latest.

[Bai+09] H. Baier et al. *QPACE – a QCD parallel computer based on Cell processors*. 11/2009. arXiv: 0911.2174 [hep-lat].

[Ber17] J. Bernier. *MIPS CPUs are at the heart of the world's greenest supercomputers*. 21/11/2017. URL: https://www.mips.com/blog/mips-cpus-are-at-the-heart-of-the-worlds-greenest-supercomputers/ (visited on 28/11/2017).

[BLAS17] *BLAS (Basic Linear Algebra Subprograms)*. 14/11/2017. URL: http://www.netlib.org/blas/ (visited on 29/11/2017).

[BM08] S. Boldo and G. Melquiond. 'Emulation of FMA and Correctly Rounded Sums: Proved Algorithms Using Rounding to Odd'. In: *IEEE Transactions on Computers* 57.4 (04/2008), pp. 462–471.

[BMP06] H. Brönnimann, G. Melquiond and S. Pion. 'The design of the Boost interval arithmetic library'. In: *Theoretical Computer Science* 351.1 (2006), pp. 111–118.

[BNS16] W. E. Brown, A. Naumann and E. Smith-Rowland. *Mathematical Special Functions for C++17, v5*. 29/02/2016. URL: https://isocpp.org/files/papers/P0226R1.pdf.

[Bou+15] A. Bouteiller et al. 'Algorithm-Based Fault Tolerance for Dense Matrix Factorizations, Multiple Failures and Accuracy'. In: *ACM Transactions on Parallel Computing* 1.2 (02-2015).

[BS08] G. Bronevetsky and B. R. de Supinski. 'Soft Error Vulnerability of Iterative Linear Algebra Methods'. In: *ICS08*. 2008.

[But+06] A. Buttari et al. 'Exploiting Mixed Precision Floating Point Hardware in Scientific Computations'. In: *High Performance Computing Workshop*. 2006.

[C++03] `C++03`. *ISO/IEC 14882:2003, Information technology – Programming languages – C++*. ISO/IEC, 2003.

[C++98] `C++98`. *ISO/IEC 14882:1998, Information technology – Programming languages – C++*. ISO/IEC, 1998.

[C99] `C99`. *ISO/IEC 9899:1999, Programming languages – C*. ISO/IEC, 1999.

[Cas08] S. Casey. *x87 and SSE Floating Point Assists in IA-32: Flush-To-Zero (FTZ) and Denormals-Are-Zero (DAZ)*. 17/10/2008. URL: https://software.intel.com/en-us/articles/x87-and-sse-floating-point-assists-in-ia-32-flush-to-zero-ftz-and-denormals-are-zero-daz/ (visited on 10/11/2017).

[Cav17] Cavium. *ThunderX ARM Processors*. 2017. URL: http://www.cavium.com/ThunderX_ARM_Processors.html (visited on 28/11/2017).

[CB17] CubieBoard. *Cubieboard 2*. 2017. URL: http://cubieboard.org/model/cb2/ (visited on 15/11/2017).

[CFD08] S. Collange, J. Flórez and D. Defour. 'A GPU interval library based on Boost interval'. In: *8th Conference on Real Numbers and Computers (RNC8)* (2008).

[Coo84] J. Coonen. 'Contributions to a proposed standard for binary floating-point arithmetic'. PhD thesis. University of California at Berkeley, 1984.

[Cro+89] J. Du Croz et al. *Validation of numerial computations in Ada*. Tech. rep. The Numerical Algorithms Group Ltd. (NAG), 1989.

[CS15] Y. Cai and S. See, eds. *GPU Computing and Applications*. Springer Singapore, 2015.

[Cuy+00] A. Cuyt et al. *The Arithmos project*. University of Antwerp (UIA). 2000. URL: http://win-www.uia.ac.be/u/cant/arithmos.

[Cuy+02] A. Cuyt et al. 'Underflow revisited'. In: *Calcolo* 39 (2002), pp. 169–179.

[Dal06] J. T. Daly. 'A higher Order estimate of the optimum checkpoint interval for restart dumps'. In: *Future Generation Computer Systems* 22 (2006), pp. 303–312.

[Dar+06] C. Daramy-Loirat et al. *CR-LIBM – A library of correctly rounded elementary functions in double-precision*. Tech. rep. LIP, 12/2006.

[Daw14] B. Dawson. *There are Only Four Billion Floats — So Test Them All!* 01/2014. URL: https://randomascii.wordpress.com/2014/01/27/theres-only-four-billion-floatsso-test-them-all/.

[DDP15] C. Denis, P. De Oliveira Castro and E. Petit. *Verificarlo: checking floating point accuracy through Monte Carlo Arithmetic*. 09/2015. arXiv: 1509.01347 [cs.MS].

[DHP15] J. Dongarra, M. A. Heroux and L. Piotr. *HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems*. Tech. rep. UT-EECS-15-736. Electrical Engineering and Computer Sciente Department, Knoxville, Tennessee, 2015.

[DIN92] *DIN 1333:1992-02, Presentation of numerical data*. 02/1992.

[Ede97] A. Edelman. 'The Mathematics Of The Pentium Bug'. In: *SIAM* 1.39 (1997), pp. 54–67.

[EHM14] J. Elliott, M. Hoemmen and F. Mueller. 'Evaluating the Impact of SDC on the GMRES Iterative Solver'. In: *IPDPS14*. 2014.

[Eib14] J. Eibl. *KDiff3*. 2014. URL: http://kdiff3.sourceforge.net (visited on 12/09/2017).

[FBS17] J. Firebaugh, O. Bruggeman and J. Snyder. *Kompare – Diff/Patch Frontend*. 2017. URL: https://www.kde.org/applications/development/kompare/ (visited on 12/09/2017).

[Fia+12] D. Fiala et al. 'Detection and Correction of Silent Data Corruption for Large-Scale High Performance Computing'. In: *SC12*. 2012.

[FL16] F. Févotte and B. Lathuilière. 'VERROU: Assessing Floating-Point Accuracy Without Recompiling'. Preprint. 10/2016. URL: https://hal.archives-ouvertes.fr/hal-01383417.

[FNZ12] A. Frommer, A. Nobile and P. Zingler. *Deflation and Flexible SAP-Preconditioning of GMRES in Lattice QCD Simulation*. 04/2012. arXiv: 1204.5463 [hep-lat].

[Fou+08] L. Fousse et al. 'MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding.' In: *ACM Transactions on Mathematical Software* 33 (06/2008), pp. 13–26.

[Fox12] T. Fox. *QPX Architecture – Quad Processing eXtension to the Power ISA*. 2012. URL: https://www.alcf.anl.gov/files/Qpx_3.pdf.

[G+10] G. Guennebaud, B. Jacob et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org.

[Gel16] J. D. Gelas. *Investigating Cavium's ThunderX: The First ARM Server SoC With Ambition*. AnandTech. 13/06/2016. URL: https://www.anandtech.com/show/10353/investigating-cavium-thunderx-48-arm-cores (visited on 28/11/2017).

[GNU16a] GNU. *An Introduction to the Autotools*. 02/06/2016. URL: http://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html (visited on 02/06/2016).

[GNU16b] GNU. *Autoconf manual*. 15/08/2016. URL: https://www.gnu.org/software/autoconf/manual/autoconf.html (visited on 15/08/2016).

[GNU16c] GNU. *The GNU Multiple Precision Arithmetic Library*. 2016. URL: http://gmplib.org (visited on 22/08/2017).

[GNU16d] GNU. *The Multiple Precision Floating-Point Reliable Library*. 3.1.5. 2016. URL: http://www.mpfr.org/mpfr-current/mpfr.pdf.

[Gol91] D. Goldberg. 'What Every Computer Scientist Should Know About Floating-Point Arithmetic'. In: *Computing Surveys* (03/1991).

[Graph500] *GRAPH 500*. URL: http://graph500.org (visited on 08/08/2017).

[Green500] *The GREEN 500 – The List*. URL: https://www.top500.org/green500/ (visited on 08/08/2017).

[GRW17] P. Georg, D. Richtmann and T. Wettig. 'DD-αAMG on QPACE 3'. In: *35th International Symposium on Lattice Field Theory (Lattice 2017) Granada, Spain, June 18-24, 2017*. 10/2017. arXiv: 1710.07041 [hep-lat].

[GV96] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd Ed.)* Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[Har00a] J. Harrison. 'Formal verification of floating-point trigonometric functions'. In: *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD 2000*. Ed. by W. A. Hunt and S. D. Johnson. Vol. 1954. Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2000, pp. 217–233.

[Har00b] J. Harrison. 'Formal verification of IA-64 divison algorithms'. In: *Proceedings of the 13th International Conference on Theorem Proving in High Order Logics, TPHOLs 2000*. Ed. by M. Aagard and J. Harrison. Vol. 1869. Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2000, pp. 234–251.

[Hau17] J. Hauser. *Berkeley SoftFloat, Release 3d*. 10/08/2017. URL: http://www.jhauser.us/arithmetic/SoftFloat.html (visited on 22/08/2017).

[HE02] R. Hain and S. Eversmeier. 'Denormalisation. Die Achillesferse des virtuellen Studios und des Pentium 4'. In: *Keyboards* (8-2002), pp. 42–45.

[HF06] M. Hüsken and A. Frommer. *Ensuring numerical quality in grid computing*. Tech. rep. BUW-SC 2006/1. University of Wuppertal, 2006.

[Hig02] N. Higham. *Accuracy and Stability of Numerical Algorithms*. Second edition. SIAM, 2002.

[Hig15] N. Higham. *The Rise of Mixed Precision Arithmetic*. 20/10/2015. URL: https://nickhigham.wordpress.com/2015/10/20/the-rise-of-mixed-precision-arithmetic/ (visited on 29/11/2017).

[Hou+88] 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.

[Hou08] D. Hough. *Revising ANSI/IEEE Std 754-1985*. 24/06/2008. URL: http://754r.ucbtest.org/web-2008 (visited on 01/12/2017).

[Hou15] D. Hough. *IEEE 754-2008 Revision Minutes, 2015 September 22*. 2015. URL: http://754r.ucbtest.org/minutes/2015-09-22-minutes.txt (visited on 01/12/2017).

[Hou17a] D. Hough. *Changes in 754-2018 from ANSI/IEEE Std 754-2008*. 21/11/2017. URL: http://754r.ucbtest.org/changes.html (visited on 29/11/2017).

[Hou17b] D. Hough. *Revising ANSI/IEEE Std 754-2008*. 30/11/2017. URL: http://754r.ucbtest.org (visited on 01/12/2017).

[HP11] J. Hennessy and D. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[HPCG] *HPCG*. URL: http://www.hpcg-benchmark.org.

[IBM01] IBM. *Workload Management with LoadLeveler*. 2001. URL: https://www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf.

[IBM03a] IBM. *Engineering and Scientific Subroutine Library for AIX, Version 4 Release 1, and Linux on pSeries, Version 4 Release 1: Guide and Reference*. 2nd edition. 2003.

[IBM03b] IBM. *Parallel Engineering and Scientific Subroutine Library for AIX, Version 3 Release 1, and Linux on pSeries, Version 3 Release 1: Guide and Reference*. 2nd edition. 2003.

[IBM11a] IBM. *IBM Blue Gene/Q*. URL: https://www-03.ibm.com/systems/technicalcomputing/solutions/bluegene/ (visited on 29/11/2017).

[IBM11b] IBM. *Icons of Progress: Blue Gene*. 2011. URL: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/bluegene/ (visited on 08/08/2017).

[IBM17] IBM Knowledge Center. *8247-42L (IBM Power System S824L)*. 14/09/2017. URL: https://www.ibm.com/support/knowledgecenter/en/8247-42L/p8hdx/8247_42l_landing.htm (visited on 20/11/2017).

[IBTA17] *InfiniBand Trade Association*. 2017. URL: http://www.infinibandta.org (visited on 23/08/2017).

[IEEE08] *IEEE Std 754-2008, Standard for Floating-Point Arithmetic*. IEEE, 2008.

[IEEE15] *IEEE Std 1788-2015, IEEE Standard for Interval Arithmetic*. IEEE, 2015.

[IEEE17a] IEEE. *IEEE 754-2008 errata*. 01/12/2017. URL: http://speleotrove.com/misc/IEEE754-errata.html (visited on 12/01/2017).

[IEEE17b] IEEE. *Revising Standards*. 2017. URL: https://standards.ieee.org/develop/revisestds.html (visited on 29/11/2017).

[IEEE85] *ANSI/IEEE Std 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, 1985.

[IEEE87] *ANSI/IEEE Std 854-1987, Standard for Radix-independent Floating-Point Arithmetic*. IEEE, 1987.

[INT03] Intel. *IA-32 Intel Architecture Software Developer's Manual*. 2003.

[INT04] Intel. *Statistical Analysis of Floating Point Flaw: Intel White Paper*. Tech. rep. 09/07/2004. URL: http://download.intel.com/support/processors/pentium/sb/FDIV_Floating_Point_Flaw_Pentium_Processor.pdf.

[INT13a] Intel ARK. *Intel Core i7-4770 Processor*. 2013. URL: https://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz (visited on 14/11/2017).

[INT13b] Intel ARK. *Intel Core i7-4800MQ Processor*. 2013. URL: https://ark.intel.com/products/75128/Intel-Core-i7-4800MQ-Processor-6M-Cache-up-to-3_70-GHz (visited on 14/11/2017).

[INT15] Intel. *Intel® Xeon Phi™ Coprocessor x100 Product Family Datasheet*. 2015. URL: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-coprocessor-datasheet.pdf.

[INT16a] Intel ARK. *Intel Xeon Phi Processor 7210*. 2016. URL: https://ark.intel.com/products/94033/Intel-Xeon-Phi-Processor-7210-16GB-1_30-GHz-64-core (visited on 14/11/2017).

[INT16b] Intel ARK. *Intel Xeon Processor E5-2620 v4*. 2016. URL: https://ark.intel.com/products/92986/Intel-Xeon-Processor-E5-2620-v4-20M-Cache-2_10-GHz (visited on 14/11/2017).

[INT16c] Intel ARK. *Intel Xeon Processor E5-2650 v4*. 2016. URL: https://ark.intel.com/products/91767/Intel-Xeon-Processor-E5-2650-v4-30M-Cache-2_20-GHz (visited on 14/11/2017).

[INT16d] Intel ARK. *Intel Xeon Processor E5-2699 v4*. 2016. URL: https://ark.intel.com/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz (visited on 14/11/2017).

[INT17a] Intel. *6th Generation Intel Processor Families for S-Platforms*. 2017. URL: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/desktop-6th-gen-core-family-datasheet-vol-1.pdf.

[INT17b] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. 2017. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[INT17c] Intel. *Intel Math Kernel Library*. 2017. URL: https://software.intel.com/en-us/mkl (visited on 29/11/2017).

[INT17d] Intel. *Intel® Xeon Phi™ Processor x200 Product Family Datasheet*. 2017. URL: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-processor-x200-product-family-datasheet.pdf.

[JSC17a] Jülich Supercomputing Center (JSC). *JUQUEEN - Jülich Blue Gene/Q*. 09/06/2017. URL: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html (visited on 14/11/2017).

[JSC17b] Jülich Supercomputing Center (JSC). *QPACE3 – Quantum Chromodynamics Parallel Computing on the Cell*. 02/08/2017. URL: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/QPACE3/_node.html (visited on 14/11/2017).

[Kah06] W. Kahan. *How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?* 2006. URL: http://people.eecs.berkeley.edu/~wkahan/Mindless.pdf.

[Kah81] W. Kahan. *Why do we need a floating-point standard?* 1981. URL: http://www.cs.berkely.edu/~wkahan/ieee754status/why-ieee.pdf.

[Kah96a] W. Kahan. *A test for correctly rounded SQRT*. 1996. URL: http://www.cs.berkely.edu/~wkahan/SQRTest.ps.

[Kah96b] W. Kahan. *Lecture notes on the status of IEEE-754*. 1996. URL: http://www.cs.berkely.edu/~wkahan/ieee754status/IEEE754.PDF.

[Kah98] W. Kahan. *The Improbability of Probabilistic Error Analyses for Numerical Computations*. 1998. URL: http://people.eecs.berkeley.edu/~wkahan/improber.pdf.

[Kar85] R. Karpinski. 'Paranoia: A floating-point benchmark'. In: *Byte* (1985), pp. 223–235.

[KD98] W. Kahan and J. Darcy. *How JAVA's Floating-Point Hurts Everyone Everywhere*. 1998. URL: http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf.

[Knu64] D. E. Knuth. 'Backus Normal Form vs. Backus Naur Form'. In: *Commununications of the ACM* 7.12 (12/1964), pp. 735–736.

[Kor+10] P. Kornerup et al. 'Computing Correctly Rounded Integer Powers in Floating-Point Arithmetic'. In: *ACM Transactions on Mathematical Software* 37.1 (01/2010).

[Kor+12] P. Kornerup et al. 'On the Computation of Correctly Rounded Sums'. In: *IEEE Transactions on Computers* 61.3 (03/2012), pp. 289–298.

[Kul89] U. Kulisch. *Wissenschaftliches Rechnen mit Ergebnisverifikation. Eine Einführung*. Akademie-Verlag, 1989.

[Lam16] T. Lampe. *Super Powerful, Super Energy-Efficient, and Super Cool – How Fujitsu is transforming High Performance Computing*. 22/12/2016. URL: http://blog.global.fujitsu.com/super-powerful-super-energy-efficient-and-super-cool-how-fujitsu-is-transforming-high-performance-computing/ (visited on 14/11/2017).

[LAP17] *LAPACK – Linear Algebra PACKage*. 14/11/2017. URL: http://www.netlib.org/lapack/ (visited on 29/11/2017).

[Lau08] C. Q. Lauter. 'Arrondi correct de fonctions mathématiques'. PhD thesis. Docteur de l'Université de Lyon - École Normale Supérieure de Lyon, 2008.

[LCJ10] J.-L. Lamotte, J.-M. Chesneaux and F. Jézéquel. 'CADNA_C: A version of CADNA for use with C or C++ programs'. In: *Computer Physics Communications* 181.11 (2010), pp. 1925–1926.

[Lef11] V. Lefèvre. 'Introduction to the GNU MPFR Library'. In: GNU Hackers Meeting. Paris, 28/08/2011. URL: http://perso.ens-lyon.fr/guillaume.hanrot/Papers/toms.pdf.

[LM01a] T. Lang and J. M. Muller. 'Bounds on Runs of Zeros and Ones for Algebraic Functions'. In: *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*. 2001, pp. 13–20.

[LM01b] V. Lefevre and J. M. Muller. 'Worst Cases for Correct Rounding of the Elementary Functions in Double Precision'. In: *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*. 2001, pp. 111–118.

[LT98] V. Lefèvre and J.-M. M. A. Tisserand. *The Table Maker's Dilemma*. Research Report 98-12. École Normale Supérieure de Lyon, 1998.

[Mel17] Mellanox. *Scalable Hierarchical Aggregation Protocol (SHARP): Release Notes*. 2017.

[MG14] M. Moldaschl and W. N. Gansterer. 'Comparison of Eigensolvers for Symmetric Band Matrices'. In: *Sci. Comput. Program.* 90 (09/2014), pp. 55–66.

[MIP17] MIPS. *IP Cores*. 2017. URL: https://www.mips.com/products/ (visited on 28/11/2017).

[MLK98] J. S. Moore, T. Lynch and M. Kaufmann. 'A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm'. In: *IEEE Transactions on Computers* 47.9 (09/1998), pp. 913–926.

[MM13] D. Moser and H. Menke. *Diffuse – graphical tool for merging and comparing text files*. 2013. URL: http://diffuse.sourceforge.net (visited on 12/09/2017).

[MMM13] É. Martin-Dorel, G. Melquiond and J.-M. Muller. 'Some issues related to double rounding'. In: *BIT Numerical Mathematics* 53.4 (2013), pp. 897–924.

[MOD17] *Environment modules*. 2017. URL: http://modules.sourceforge.net (visited on 29/11/2017).

[Moo+10] A. Moody et al. 'Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System'. In: *SC10*. 2010.

[Moo66] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[Mor17a] T. P. Morgan. *A Deep Dive Into NEC's Aurora Vector Engine*. The Next Platform. 22/11/2017. URL: https://www.nextplatform.com/2017/11/22/deep-dive-necs-aurora-vector-engine/ (visited on 28/11/2017).

[Mor17b] T. P. Morgan. *Qualcomm's Amberwing Arm Server Chip Finally Takes Flight*. The Next Platform. 08/11/2017. URL: https://www.nextplatform.com/2017/11/08/qualcomms-amberwing-arm-server-chip-finally-takes-flight/ (visited on 28/11/2017).

[Mor17c] T. P. Morgan. *The Power9 Rollout Begins With Summit And Sierra Supercomputers*. The Next Platform. 19/09/2017. URL: https://www.nextplatform.com/2017/09/19/power9-rollout-begins-summit-sierra/ (visited on 28/11/2017).

[MPFR16] GNU. *The GNU MPFR Library*. 2016. URL: http://www.mpfr.org (visited on 29/11/2017).

[MPI15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. 04/06/2015. URL: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[Mue+05] S. M. Mueller et al. 'The Vector Floating-Point Unit in a Synergistic Processor Element of a CELL Processor'. In: *ARITH'05*. 06/2005, pp. 59–67.

[Mül+07] R. Müller-Pfefferkorn et al. 'User-Centric Monitoring and Steering of the Execution of Large Job Sets'. In: *Conference Proceedings German e-Science Conference*. 2007.

[Mul+10] J.-M. Muller et al. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[NEC17] NEC. *NEC releases new high-end HPC product line, SX-Aurora TSUBASA*. 25/10/2017. URL: http://www.nec.com/en/press/201710/global_20171025_01.html (visited on 28/11/2017).

[NG13] G. Niederbrucker and W. N. Gangsterer. 'Robust Gossip-Based Aggregation: A Practical Point Of View'. In: *ALENEX13*. 2013, pp. 133–147.

[Nic11] T. R. Nicely. *Pentium FDIV flaw*. 2011. URL: http://www.trnicely.net/pentbug/pentbug.html (visited on 19/08/2011).

[Nob11] A. Nobile. *Solving the Dirac equation on QPACE*. 09/2011. arXiv: 1109.4279 [hep-lat].

[Num17] NumPy. *Numerical Python – A package for scientific computing with Python*. 2017. URL: http://www.numpy.org/ (visited on 29/11/2017).

[NVi16a] NVidia. *NVIDIA Tesla P100. GP100 Pascal Whitepaper*. Vol. WP-08019-001_v01.1. 2016.

[NVi16b] NVidia. *QUADRO P6000 Datasheet*. 12/09/2016. URL: http://images.nvidia.com/content/pdf/quadro/data-sheets/192152-NV-DS-Quadro-P6000-US-12Sept-NV-FNL-WEB.pdf (visited on 22/11/2017).

[NVi17a] NVidia. *CUDA C Programming Guide*. Vol. PG-02829-001_v8.0. 2017.

[NVi17b] NVidia. *NVidia Volta*. 2017. URL: https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/ (visited on 28/11/2017).

[Ora16a] Oracle. *Java Native Interface*. 2016. URL: https://docs.oracle.com/javase/6/docs/technotes/guides/jni/ (visited on 23/08/2017).

[Ora16b] Oracle. *Java Platform Standard Ed. 8, Class Math*. 2016. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html (visited on 23/08/2017).

[Par97] D. S. Parker. *Monte Carlo Arithmetic: exploiting randomness in floating-point arithmetic*. Tech. rep. CSD-970002. University of California, Los Angeles, Computer Science Department, 1997. URL: http://web.cs.ucla.edu/~stott/mca/CSD-970002.ps.gz.

[PBS16] Altair PBS Works. *PBS Professional 13.1 User's Guide*. 2016. URL: http://www.pbsworks.com/pdfs/PBSProUserGuide13.1.pdf.

[Pet+16] A. Petitet et al. *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. 2016. URL: http://www.netlib.org/benchmark/hpl/.

[Pin17] Pine64. *64-bit Single Board Computer: Simple, Expandable & Powerful*. 2017. URL: https://www.pine64.org/?page_id=1194 (visited on 17/11/2017).

[PK91] V. Paxson and W. Kahan. *A program for testing IEEE decimal-binary conversion*. Tech. rep. University of California at Berkely, USA, 1991.

[Qua17] Qualcomm. *Qualcomm Centriq 2400 – the world's first 10nm server processor*. 08/11/2017. URL: https://www.qualcomm.com/news/onq/2017/11/08/qualcomm-centriq-2400-worlds-first-10nm-server-processor (visited on 28/11/2017).

[Res16] ResearchGate. *MPFR: In general, what should the values of e_{max} and e_{min} be to get correct subnormal numbers in different precisions?* 31/06/2016. URL: https://www.researchgate.net/post/MPFR_In_general_what_should_the_values_of_e_max_and_e_min_be_to_get_correct_subnormal_numbers_in_different_precisions (visited on 24/08/2017).

[RPI17a] Raspberry Pi Foundation. *BCM2835*. 2017. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md (visited on 15/11/2017).

[RPI17b] Raspberry Pi Foundation. *BCM2837*. 2017. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md (visited on 15/11/2017).

[RPI17c] Raspberry Pi Foundation. *Raspberry Pi*. 2017. URL: https://www.raspberrypi.org (visited on 15/11/2017).

[Rum13a] S. M. Rump. 'Accurate solution of dense linear systems, part I: Algorithms in rounding to nearest'. In: *Journal of Computational and Applied Mathematics* 242 (2013), pp. 157–184.

[Rum13b] S. M. Rump. 'Accurate solution of dense linear systems, Part II: Algorithms using directed rounding'. In: *Journal of Computational and Applied Mathematics* 242 (2013), pp. 185–212.

[Rus00] D. M. Russinoff. 'A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor'. In: *Lecture Notes in Computer Science*. Lecture Notes in Computer Science 1954 (2000). Ed. by W. A. Hunt and S. D. Johnson, pp. 3–36.

[Rus98] D. M. Russinoff. 'A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions'. In: *LMS Journal of Computation and Mathematics*. Lecture Notes in Computer Science 1 (1998). Ed. by W. A. Hunt and S. D. Johnson, pp. 148–200.

[Rus99] D. M. Russinoff. 'A mechanically checked proof of correctness of the AMD K5 floating point square root microcode'. In: *Formal Methods in System Design*. Lecture Notes in Computer Science 14.1 (1999). Ed. by W. A. Hunt and S. D. Johnson, pp. 75–125.

[Sad+17] S. K. Sadasivam et al. 'IBM Power9 Processor Architecture'. In: *IEEE Micro* 37.2 (03/2017), pp. 40–51.

[Sch17] D. Schor. *The 2,048-core PEZY-SC2 sets a Green500 record*. WikiChip. 01/11/2017. URL: https://fuse.wikichip.org/news/191/the-2048-core-pezy-sc2-sets-a-green500-record/ (visited on 28/11/2017).

[Sco85] N. R. Scott. *Computer Number Systems and Arithmetic*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[SFBTR] SFB/TR55. *Hadron Physics from Lattice QCD*. URL: http://www.physik.uni-regensburg.de/sfbtr55/ (visited on 28/11/2017).

[Shi13] A. L. Shimpi. *The ARM Diaries, Part 1: How ARM's Business Model Works*. AnandTech. 28/06/2013. URL: http://www.anandtech.com/show/7112/the-arm-diaries-part-1-how-arms-business-model-works (visited on 28/11/2017).

[Shi17] A. Shilov. *Intel Discontinues Xeon Phi 7200-Series 'Knights Landing' Coprocessor Cards*. AnandTech. 25/08/2017. URL: http://www.anandtech.com/show/11769/intel-discontinues-xeon-phi-7200-series-knights-landing-coprocessor-cards (visited on 28/11/2017).

[SK11] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1. print. Addison-Wesley, 2011.

[Smi16] R. Smith. *Intel Announces Knights Mill: A Xeon Phi For Deep Learning*. AnandTech. 17/08/2016. URL: https://www.anandtech.com/show/10575/intel-announces-knights-mill-a-xeon-phi-for-deep-learning (visited on 28/11/2017).

[Smi91] D. Smith. 'Algorithm 693: a Fortran package for floating-point multiple-precision arithmetic'. In: *ACM Transactions on Mathematical Software* 17.2 (1991), pp. 273–283. URL: http://www.lmu.edu/acad/personal/faculty/dmsmith2/FMLIB.html.

[Sni+14] M. Snir et al. 'Addressing failures in exascale computing'. In: *The International Journal of High-Performance Computing Applications* 28.2 (2014), pp. 129–173.

[SQL16a] SQLite. *An embedded SQL database engine*. 08/07/2016. URL: https://www.sqlite.org (visited on 08/07/2016).

[SQL16b] SQLite. *Most Widely Deployed and Used Database Engine*. 2016. URL: https://www.sqlite.org/mostdeployed.html (visited on 08/11/2017).

[SSL16] OpenSSL. *Cryptography and SSL/TLS Toolkit*. 16/06/2016. URL: https://openssl.org (visited on 16/06/2016).

[Sta06] G. Staples. 'TORQUE Resource Manager'. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0.

[SUN97] SUN Microsystems. *The UltraSparc processor*. Technology White Paper. SUN Microsystems, Inc., 1997.

[TOP500a] Top 500. *Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway*. URL: https://www.top500.org/system/178764 (visited on 01/12/2017).

[TOP500b] *Top 500 – The List*. URL: https://www.top500.org (visited on 08/08/2017).

[VAL17] Valgrind. *An instrumentation framework for building dynamic analysis tools*. 2017. URL: http://valgrind.org (visited on 16/11/2017).

[VCV01a] B. Verdonk, A. Cuyt and D. Verschaeren. 'A precision- and range-independent tool for testing floating-point arithmetic I: basic operations, square root and remainder'. In: *ACM Transactions on Mathematical Software* 27 (2001), pp. 92–118.

[VCV01b] B. Verdonk, A. Cuyt and D. Verschaeren. 'A precision- and range-independent tool for testing floating-point arithmetic II: conversions'. In: *ACM Transactions on Mathematical Software* 27 (2001), pp. 119–140.

[Vig93] J. Vignes. 'A stochastic arithmetic for reliable scientific computation'. In: *Mathematics and Computers in Simulation* 35.3 (1993), pp. 233–261.

[WIK16] Wikipedia. *IEEE 754 Revision.* 28/05/2016. URL: https://en.wikipedia.org/wiki/IEEE_754_revision (visited on 03/11/2017).

[WIK17a] Wikipedia. *3DNow!* 01/10/2017. URL: https://en.wikipedia.org/wiki/3DNow! (visited on 03/11/2017).

[WIK17b] Wikipedia. *Advanced Vector Extensions*. 21/10/2017. URL: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions (visited on 03/11/2017).

[WIK17c] Wikipedia. *ARM architecture*. 31/10/2017. URL: https://en.wikipedia.org/wiki/ARM_architecture (visited on 03/11/2017).

[WIK17d] Wikipedia. *AVX-512*. 29/10/2017. URL: https://en.wikipedia.org/wiki/AVX-512 (visited on 03/11/2017).

[WIK17e] Wikipedia. *Blue Gene*. 13/10/2017. URL: https://en.wikipedia.org/wiki/Blue_Gene (visited on 03/11/2017).

[WIK17f] Wikipedia. *Cell (microprocessor)*. 08/09/2017. URL: https://en.wikipedia.org/wiki/Cell_(microprocessor) (visited on 03/11/2017).

[WIK17g] Wikipedia. *Criticism of Java*. 02/09/2017. URL: https://en.wikipedia.org/wiki/Criticism_of_Java#Floating_point_arithmetic (visited on 03/11/2017).

[WIK17h] Wikipedia. *CUDA*. 24/10/2017. URL: https://en.wikipedia.org/wiki/CUDA (visited on 03/11/2017).

[WIK17i] Wikipedia. *Distributed computing*. 02/11/2017. URL: https://en.wikipedia.org/wiki/Distributed_computing (visited on 03/11/2017).

[WIK17j] Wikipedia. *Endianess*. 29/09/2017. URL: https://en.wikipedia.org/wiki/Endianess (visited on 03/11/2017).

[WIK17k] Wikipedia. *Environment Modules (software)*. 20/09/2017. URL: https://en.wikipedia.org/wiki/Environment_Modules_(software) (visited on 03/11/2017).

[WIK17l] Wikipedia. *Field-programmable gate array*. 25/10/2017. URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array (visited on 03/11/2017).

[WIK17m] Wikipedia. *Floating-point arithmetic*. 27/10/2017. URL: https://en.wikipedia.org/wiki/Floating-point_arithmetic#Rounding_modes (visited on 03/11/2017).

[WIK17n] Wikipedia. *General-purpose computing on graphics processing units*. 30/10/2017. URL: https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units (visited on 03/11/2017).

[WIK17o] Wikipedia. *GNU Build System*. 30/10/2017. URL: https://en.wikipedia.org/wiki/GNU_Build_System (visited on 03/11/2017).

[WIK17p] Wikipedia. *IBM Roadrunner*. 14/06/2017. URL: https://en.wikipedia.org/wiki/IBM_Roadrunner (visited on 03/11/2017).

[WIK17q] Wikipedia. *IEEE floating point*. 28/10/2017. URL: https://en.wikipedia.org/wiki/IEEE_floating_point (visited on 03/11/2017).

[WIK17r] Wikipedia. *Interval arithmetic*. 29/08/2017. URL: https://en.wikipedia.org/wiki/Interval_arithmetic (visited on 03/11/2017).

[WIK17s] Wikipedia. *Java (programming language)*. 08/10/2017. URL: https://en.wikipedia.org/wiki/Java_(programming_language) (visited on 03/11/2017).

[WIK17t] Wikipedia. *List of numerical libraries*. 02/11/2017. URL: https://en.wikipedia.org/wiki/List_of_numerical_libraries (visited on 03/11/2017).

[WIK17u] Wikipedia. *OpenCL*. 10/10/2017. URL: https://en.wikipedia.org/wiki/OpenCL (visited on 03/11/2017).

[WIK17v] Wikipedia. *Power Architecture*. 06/09/2017. URL: https://en.wikipedia.org/wiki/Power_Architecture (visited on 03/11/2017).

[WIK17w] Wikipedia. *Power.org*. 26/08/2017. URL: https://en.wikipedia.org/wiki/Power.org (visited on 03/11/2017).

[WIK17x] Wikipedia. *QPACE*. 29/07/2017. URL: https://en.wikipedia.org/wiki/QPACE (visited on 03/11/2017).

[WIK17y] Wikipedia. *Quantum chromodynamics*. 25/10/2017. URL: https://en.wikipedia.org/wiki/Quantum_chromodynamics (visited on 03/11/2017).

[WIK17z] Wikipedia. *Single-board computer*. 03/08/2017. URL: https://en.wikipedia.org/wiki/Single-board_computer (visited on 03/11/2017).

[WIK17aa] Wikipedia. *SSE2*. 03/09/2017. URL: https://en.wikipedia.org/wiki/SSE2 (visited on 03/11/2017).

[WIK17ab] Wikipedia. *SSE3*. 13/07/2017. URL: https://en.wikipedia.org/wiki/SSE3 (visited on 03/11/2017).

[WIK17ac] Wikipedia. *SSE4*. 22/07/2017. URL: https://en.wikipedia.org/wiki/SSE4 (visited on 03/11/2017).

[WIK17ad] Wikipedia. *SSSE3*. 26/09/2017. URL: https://en.wikipedia.org/wiki/SSSE3 (visited on 03/11/2017).

[WIK17ae] Wikipedia. *Streaming SIMD Extensions*. 01/09/2017. URL: https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions (visited on 03/11/2017).

[WIK17af] Wikipedia. *Sunway*. 24/11/2017. URL: https://en.wikipedia.org/wiki/Sunway (visited on 29/11/2017).

[WIK17ag] Wikipedia. *Sunway TaihuLight*. 28/11/2017. URL: https://en.wikipedia.org/wiki/Sunway_TaihuLight (visited on 29/11/2017).

[WIK17ah] Wikipedia. *SW26010*. 02/10/2017. URL: https://en.wikipedia.org/wiki/SW26010 (visited on 29/11/2017).

[WIK17ai] Wikipedia. *VEX prefix*. 15/06/2017. URL: https://en.wikipedia.org/wiki/VEX_prefix (visited on 03/11/2017).

[WIK17aj] Wikipedia. *x86*. 18/10/2017. URL: https://en.wikipedia.org/wiki/X86 (visited on 03/11/2017).

[WIK17ak] Wikipedia. *x87*. 26/08/2017. URL: https://en.wikipedia.org/wiki/X87 (visited on 03/11/2017).

[Wil12] K. Willadsen. *Meld*. 2012. URL: http://meldmerge.org (visited on 12/09/2017).

[WLCG] WLCG. *Worldwide LHC Computing Grid*. URL: http://wlcg.web.cern.ch (visited on 29/11/2017).

[YJG03] A. B. Yoo, M. A. Jette and M. Grondona. 'SLURM: Simple Linux Utility for Resource Management'. In: *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper*. Ed. by D. Feitelson, L. Rudolph and U. Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.