Implementation and Verification of IEEE-conform Floating Point Arithmetic

Jacqueline Stratmann

MASTERTHESIS

Department of Computer Science
University of Kaiserslautern

November 2015

Supervisors:

Prof. Dr. Klaus Schneider
Maximilian Senftleben

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mir beim Verfassen dieser Arbeit geholfen haben, in erster Linie meinem Betreuer Prof. Dr. Klaus Schneider.

Ganz besonders danken möchte ich außerdem meinen Eltern, meinen Schwestern Julie und Fabienne und meinem Lebensgefährten Daniel. Ihr habt mich immer unterstützt und wart jederzeit für mich da. Vielen Dank, dass es Euch gibt und dass ihr Teil meines Lebens seid!

Erklärung / Statement

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

I declare, that this thesis and the code has been composed by myself, that I did not use other sources than the given ones and that I marked all passages which were taken from other sources.

Kaiserslautern, 03.11.2015

Jacqueline Stratmann

Zusammenfassung / Abstract

Diese Abschlussarbeit behandelt die Definition, Verifikation und Implementierung einer IEEE-konformen Floating Point Arithmetik für binäre Floating Point Zahlen. Da sich Floating Point Zahlen sowohl für die Darstellung sehr kleiner, als auch sehr großer Zahlen eignen, werden sie heutzutage in nahezu jedem Computer benutzt. Natürlich ist es wünschenswert, dass Ergebnisse von verschiedenen Implementierungen vergleichbar sind und sie bei den gleichen Eingaben zum gleichen Ergebnis kommen. Dafür muss klar sein, welche Anforderungen an eine Implementierung gestellt werden und wann ein Ergebnis korrekt ist. Der IEEE Standard 754 bietet grundsätzlich genau das. Da aber nur eine textuelle Beschreibung für die Definitionen gewählt wurde und auch nicht alle Sonderfälle abgedeckt sind, ergeben sich einige Freiheiten für Implementierungen, die sich an den Standard halten. Eine zusätzliche, vollständige Definition ist also notwendig.
In dieser Arbeit wird zuerst das Format erklärt. Außerdem werden die Exceptions und Spezialfälle kurz erläutert. Danach folgt eine Auflistung aller erforderlichen Operationen mit kurzer Beschreibung.
Die zuvor genannten Operationen werden dann definiert, wofür hauptsächlich Flußdiagramme als Darstellungsform gewählt wurden.
Im Anschluss an die Definitionen folgt die Verifikation. Einige Operationen erfordern keine ausführliche, formale Verifikation, die Korrektheit aller anderen wird bewiesen. Zusätzlich wurden die Operationen in der Programmiersprache Quartz implementiert.

This thesis considers the definition, verification and implementation of an IEEE-conform floating point arithmetic using binäry floating point numbers. Due to the fact that floating point numbers are able to represent both very small and very large numbers, they are used in nearly every computer today. Preferable, results from different implementations should be comparable and identical inputs should lead to similar outputs. To achieve that, the requirements on such an implementation should be distinctly defined. The IEEE Standard 754 fulfills exactly that task. But the definitions are only described textual and special cases are not completely covered, so there is a certain freedom left for implementations that fulfill the requirements of the IEEE Standard. Thus an additional, complete definition is required.
First of all, the format is clarified in this thesis. Furthermore, the exceptions and special cases are briefly explained. After this, a list of all mandatory operations together with a small description follows.
The previously listet operations are then defined, mainly via flowcharts and a textual description of those.
Thereafter the definition follows. Some operations do not require a detailed, formal verification. The correctness of all others is proven. Additionally the operations were implemented usind the programming language Quartz.

Contents

List of Figures

List of Tables

1 Acronyms

ACM	Association for Computing Machinery
FP	Floating Point Number
IBM	International Business Machines Corporation
IEEE	Institute of Electrical and Electronics Engineers
IFF	If And Only If
FPGA	Field Programmable Gate Array
G	Guard Bit
GPU	Global Processing Unit
LOG	Logarithm
LSB	Least Significant Bit
MSB	Most Siginificant Bit
NaN	Not a Number
qNaN	Quiet Not a Number
R	Round Bit
S	Sticky Bit
SAL	Shift Arithmetic Left
SAR	Shift Arithmetic Right
sNaN	Signaling Not a Number
SQRT	Square Root

2 Introduction

2.1 Motivation

Fixed point numbers are very restricted because the representation of very large or very small numbers require plenty of bits, so that they are unfeasible without manual scaling. Floating point numbers offer exactly that scaling and it is used in science for a long time, e. g. for writing the speed of light: $2.9979 \cdot 10^8 \frac{m}{s}$ [5]. So they are used in nearly every computer today and because results should be comparable and reproducible, the Institute of Electrical and Electronics Engineers published the IEEE Standard 754.
Even if there are some processors and computer languages that do not comply with that standard, like some IBM mainframes, the VAX architecture or the supercomputers from Cray, or are only obedient in parts like Java [7, 13], most of the currently used processors and programming languages comply with it. But the IEEE Standard just describes the operations in a textual form, not in formal and unambiguous definitions. So the programmer will very likely have some unanswered questions after consulting the IEEE document. In many cases, the desired result is defined, but the way to get there is left in limbo. The freedom may ease implementations, but it also poses the risk, that results of different implementations are neither comparable nor reproducible.

> *"It is hoped that language-defined methods for the control of expression evaluation and exceptions might be defined in coming years, so that it will be possible to write programs that produce identical results on all conforming systems."* [11]

The quotation above is taken from the IEEE Standard 754-2008. It shows that the authors are aware of the lack of information and are desirous of closing that gap. This thesis claims to do exactly that: closing the gap between textual description and formal definition. It defines all operations that are mandatory for an IEEE conform implementation, verifies these definitions and finally offers an implementation of them.

2.2 Contribution

This thesis presents a complete definition of the operations that are mandatory for an IEEE-conform floating point arithmetic, the verification of the defined operations and their implementation. After a side trip to the history of floating point numbers used in computer science, a short introduction and a brief description of the operations and special cases is provided. These operations are accurately defined using flowcharts. The special cases and specific requirements prescribed by the IEEE Standard 754-2008 are certainly considered in these definitions. Thereafter, the defined implementations are, if the correctness can not trivially be

displayed, proven in terms of a formal verification. An implementation of the operations was done subsequently. This thesis does not elaborate on the implementation, because essentially the implemented operations equate the definitions in code style instead of flowcharts. Even so the adaptions, that were necessary, are briefly described. These implementations are tested to ensure the correctness of the implementation and double-check the definitions.

2.3 Related Work

As far as I know, there is no completely defined and verified IEEE-conform floating point arithmetic published up to the present day. Most of the publications describe the format and usually some basic arithmetic operations. Michael L. Overton presents the format, the basic operations addition, subtraction, multiplication, division and square root, rounding and exceptions in "Numerical Computing with IEEE Floating Point Arithmetic" [19]. Additionally, he briefly describes the implementation of floating point numbers in the Intel Microprocessors and the programming language C. Another publication that offers a definition of operations is "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs" [16] from Loucas Louca, Todd A. Cook and William H. Johnson. They define algorithms for addition and multiplication of single precision floating point numbers and their implementation for field programmable gate arrays (FPGAs). The algorithms are defined, examples are reckoned and the implementations are analyzed with respect to their timing and FPGA specific requirements and characteristics. Even if the title suggests IEEE-conform algorithms, the rounding is actually nonstandard.
A definition that is at least partially proven to be correct is published by David Goldberg in "What Every Computer Scientist Should Know About Floating-point Arithmetic" [6]. He provides an introduction into the floating point format, the representation and rounding errors. After that, the four basis arithmetic operations addition, subtraction, multiplication and division are canvassed and some aspects that have to be considered during the design of computer systems are discussed. Most of the statements that were made in the paper, are formally proven afterwards. But the definition is neither complete, in terms of a complete IEEE-conform arithmetic with all mandatory operations, nor completely proven.
The reference that was used most frequently while writing this thesis, beside the IEEE Standard 754-2008, is "Handbook of Floating-Point Arithmetic" [18] from Jean-Michel Muller et al.. After some historical introduction and the common informations about the format, rounding methods and exceptions, a detailed presentation of all published IEEE Standards from 754-1985 over 754-1987 to 754-2008 follows. The reason, why a revision was necessary is also explained in common with the current implementations used in processors and global processing units (GPUs). All operations that are mentioned in the IEEE Standard 754 are subsequently described, unfortunately in some cases without a definition that is clear and comprehensible and the proof of correctness is missing, too.
Beside the publications that cover the basic operations, some papers focus on special operations. Oriol Vinyals et al. in "Revisiting a basic function on current CPUs: A fast logarithm implementation with adjustable accuracy" [26] describe an algorithm that calculates the log-

arithm of a given floating point number with adjustable accuracy. That approach is used to calculate logarithms in this thesis.
The main publication this thesis is based on is the IEEE Standard 754-2008 [11], the revision of the IEEE Standard 754-1985 [10]. It textually standardizes the format, the operations, special cases et cetera that make a definition or implementation IEEE-conform.

2.4 Organization

This thesis is organized as follows:
Chapter 1 provides a list of the used acronyms and Chapter 2 presents the motivation to write this thesis and the related work.
After the preamble, Chapter 3 starts with in small trip to the history of floating point numbers used in computer science starting from Konrad Zuses Z1 and ending nowadays. The IEEE Standard 754-2008 is outlined afterwards, describing the format and special cases defined in that document and listing the operations that are mandatory for an IEEE-conform arithmetic.
The main part of this thesis starts with Chapter 4, what includes the definition of all operations mentioned in Chapter 3. The definitions are almost always done using flowcharts beside a textual explanation. Flowcharts are only relinquished if the particular operation can be defined in one sentence or a mathematical definition is more readable.
The verification of the defined operations follows in Chapter 5. The operations that do not need a formal verification, e. g. flag operations, are omited. The defined operations were implemented, Chapter 6 goes into that part of the workflow.
Finally, Chapter 7 concludes the thesis, shows the status quo of the project and proposes the future work for the time after delivering this masterthesis.

3 Preliminaries

3.1 History of Floating Point Numbers

The first computer that uses floating point numbers is the Z1 built in 1938 by Konrad Zuse [22], that uses binary floating point numbers with 24 bit, 7 bit used for the exponent, 16 bit used for the mantissa and one bit used for the sign. The further developed format used in the Z3 [22] from 1941 offers representations for signed infinities and operations with these. The Z4 [1, 4] from 1945 finally offered binary floating point numbers with 32 bit, the four basic arithmetic operations (addition, subtraction, multiplication and division), the calculation of the square root and some special operations.
While the computers built by Konrad Zuse require decimal floating point numbers as inputs and outputs, but internally work with binary ones, the Mark V [4] built in 1946 by Bell Laboratories works on decimal floating point numbers.
The first machine that makes use of a biased exponent is the IBM 704 [17] from 1954 manufactured by the International Business Machines Corporation (IBM). It is also the first mass-produced computer that uses floating point arithmetic hardware.
In 1964 IBM introduced the IBM System/360 mainframe that uses hexadecimal floating point numbers with 32 and 64 bit. This format is currently still in use in the z/Architecture [12] introduced in 2000.
William Kahan starts defining a standard for computer arithmetic in 1977 and starts publishing drafts from 1979 on [2,3,14]. In 1985, the IEEE Standard 754-1985 [10] based on his work was published. His work in creating the IEEE Standard 754 was honored in 1989 with the ACM's Touring Award. In the following years he continued to work in improving the standard and leaded the creation of the revised IEEE Standard 754-2008 [11] from 2000 till 2008 that was published in 2008.
With respect to the IEEE Standard 754-1985, IBM includes an IEEE-compatible binary floating point arithmetic in their mainframes in 1998 [9] and an IEEE-compatible decimal floating point arithmetic in 2005.

3.2 IEEE Standard 754-2008

3.2.1 Format

The IEEE Standard 754-2008 [11] defines floating point numbers conglomerated of three parts: the sign, the exponent and the mantissa. Typical formats are single precision floating point numbers built up of 32 bits or double precision floating point numbers built up of 64 bits, except for the sign, that is always represented by one bit, exponent and mantissa can have any

length. Figure 3.1 shows the segmentation of a single precision floating point number with e=8 bits for the exponent and m=23 bits for the mantissa. Figure 3.2 shows the segmentation of a double precision floating point number with e=11 bits for the exponent and m=52 bits for the mantissa.

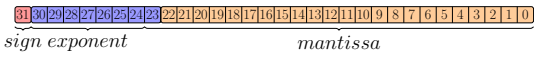

Figure 3.1: Single Precision Floating Point Format

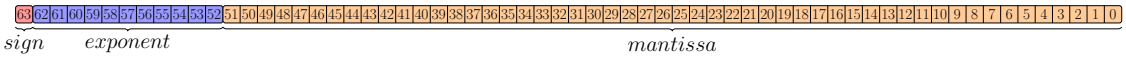

Figure 3.2: Double Precision Floating Point Format

Beside special cases, the first bit of the mantissa is a one. Due to the fact that this information is known, it does not has to be stored explicitly. This implicitly stored bit is called hidden bit. Furthermore, there is a constant that is used to implement negative exponents: the bias $\beta := (B^e - 1)\ div\ 2$. It is subtracted from the exponent, thus the largest possible value of the exponent is $e_{max} := B^{e-1} - 1 = \beta$ while the smallest possible one is $e_{min} := 1 - e_{max} = 2 - B^{e-1}$ [10, 11].
The decimal value of a normal floating point number, identifiable throught a non-zero exponent representation that is not exceeding B^{2^e-2}, is calculated as shown in Figure 3.3.

$$\langle [x_n, ..., x_0] \rangle^{\mathbb{R}}_{B,m,e} = (-1)^{x_n} \cdot (\langle [x_{m-1}, ..., x_0] \rangle^{\mathbb{N}}_{B} \cdot B^{1-m}) \cdot B^{\langle [x_{n-1}, ..., x_m] \rangle^{\mathbb{N}}_{B} - \beta}$$

Figure 3.3: Conversion of Floating Point Numbers Into Decimal Numbers [21]

3.2.2 Special cases and Exceptions

3.2.2.1 NaN, Infinity, Zero, Subnormal numbers

To represent special floating point numbers, there are several particular values for exponent and mantissa that are interpreted differently than shown in Figure 3.3 and described in Section 3.2.1. Table 3.1 depicts all encodings of special and non-special floating point numbers. The length of the mantissa is called m, the length of the exponent is called e while the value of the mantissa is called m_x, the value of the exponent is called e_x and the sign of x is called s_x.

NaN, the abbreviation for not a number, is used to mark the occurence of a undesirable problem like invalid or unavailable inputs or results. There are two kinds of NaNs, the quiet NaN (qNaN) and the signaling NaN (sNaN). Typically, qNaNs are used unless it is explicitly defined differently. By default, sNaNs are only used for signaling the invalid operation exception and the operations that use them explicitly, like *compareSignalingLessUnordered* and

exponent	mantissa	value
$2^e - 1$	$\neq 0$	NaN
$2^e - 1$	0	$(-1)^{s_x} \cdot \infty$
0	0	$(-1)^{s_x} \cdot 0$
0	$\neq 0$	$(-1)^{s_x} \cdot B^{e_{min}} \cdot (B^{1-m} \cdot m_x)$ (subnormal)
$1 \leq e_x \leq 2^e - 2$		$(-1)^{s_x} \cdot B^{e_x - \beta} \cdot (1 + B^{1-m} \cdot m_x)$ (normal)

Table 3.1: Special Floating Point Numbers and Their Representation

compareSignalingGreaterUnordered.

There exist two different infinities: $-\infty$ and $+\infty$. Beside arithmetic operations that lead to $\pm\infty$ as result, e. g. $addition(x, \infty) = \infty$ for finite x, ∞ essentially is the result of overflows. Depending on the selected rounding method, they are handled as shown in Table 3.2.

rounding method	overflow negative sign	overflow positive sign
$roundTiesToEven$	$-\infty$	$+\infty$
$roundTiesToAway$	$-\infty$	$+\infty$
$roundTowardZero$	$-\ largest\ finite\ number$	$+\ largest\ finite\ number$
$roundTowardNegative$	$-\infty$	$+\ largest\ finite\ number$
$roundTowardPositive$	$-\ largest\ finite\ number$	$+\infty$

Table 3.2: Handling of Overflows Depending on Rounding Method and Sign

Zero is basically a special subnormal number. But it is handled separately because in most operations the sign of zero does not matter, i. e. $-0 = +0$ in the majority of cases. But as always there is no rule without exception, e.g. $sqrt(-0) = -0$ while $sqrt(+0) = +0$ according to the IEEE Standard 754-2008 [11].

Subnormal numbers, also called denormal or denormalized numbers, are used to fill the gap between zero and the normal numbers with the least absolute value. There are two differences between normal and subnormal numbers. First of all the hidden bit is zero, not one. And secondly the bias is not subtracted from the exponent, the exponent is always equal to $e_{min} = 1 - e_{max} = 2 - B^{e_1}$.

3.2.2.2 Exceptions

Five exceptions are specified in the IEEE Standard 754-2008 [11]: invalid operation exception, divideByZero exception, overflow exception, underflow exception and inexact exception. They are realized via status flags that can be raised and lowered by the occurence of the corresponding exception and by the user using the flag operations defined in Section 4.2.9.

The five exceptions are described in more detail in the following.

invalid operation exception

It is raised if the result is undefined. The IEEE Standard specifies the following groups of operations that lead to an invalid operation exception:

- operations on a $sNaN$ except some conversion operations
- $multiplication(0, \infty)$ and $multiplication(\infty, 0)$
- $fusedMultiplyAdd(0, \infty, c)$ and $fusedMultiplyAdd(\infty, 0, c)$ with c being no $qNaN$
- $addition(+\infty, -\infty)$, $addition(-\infty, +\infty)$, $subtraction(+\infty, +\infty)$, $subtraction(-\infty, -\infty)$ and $fusedMultiplyAdd$ that lead to magnitude subtraction of infinities
- $division(0, 0)$ and $division(\infty, \infty)$
- $remainder(\pm\infty, c)$ and $remainder(c, \pm 0)$ with c being no NaN
- $sqrt(x)$ with $x < 0$
- result is too large or too small to be representable
- conversion from floating point to integer of a NaN, $\pm\infty$ or an input that leads to an overflow
- unordered comparisons according to their definition
- $logB(NaN)$, $logB(\infty)$ and $logB(0)$ for integer formats

divideByZero exception

It is raised if one of the two following cases fit:

- $division(x, +0) = (-1)^{s_x} \cdot \infty$ and $division(x, -0) = (-1)^{\neg s_x} \cdot \infty$ with $x \neq 0$ being a finite floating point number, the result
- $logB(0) = -\infty$ for floating point formatis

overflow exception

It is raised if an overflow occurs, which means that the absolute value of the result is too large to be representable. The handling of overflows is defined in Table 3.2.

underflow exception

It is raised if an underflow occurs, which means that the absolute value of the non-zero result is too small to be representable.

inexact exception

Is it raised if the result of an operation has to be rounded to be representable, i. e. the returned result is not equal to the result that would be returned with unbounded accuracy.

3.2.3 Operations

All operations that are compulsory for an IEEE conform floating point arithmetic and that are therefore defined, verified and implemented in the course of writing this thesis, are briefly described in this subsection. The grouping is taken from the IEEE Standard 754-2008 [11] and retained during the thesis.

3.2.3.1 General computational operations

sourceFormat* roundToIntegralTiesToEven *(source0)
The input is rounded towards the next integral value, if the difference to two integral values is equal, the even value is chosen as result.

sourceFormat* roundToIntegralTiesToAway *(source0)
The input is rounded towards the next integral value, if the difference to two integral values is equal, the input value is rounded away from zero.

sourceFormat* roundToIntegralTowardZero *(source0)
If the input is not an integral value, it is rounded towards zero, no matter if that is the nearest integral value.

sourceFormat* roundToIntegralTowardPositive *(source0)
If the input is not an integral value, it is rounded towards $+\infty$, no matter if that is the nearest integral value.

sourceFormat* roundToIntegralTowardNegative *(source0)
If the input is not an integral value, it is rounded towards $-\infty$, no matter if that is the nearest integral value.

sourceFormat* roundToIntegralExact *(source0)
If the input is NaN, an invalid operand exception is raised. Otherwise, the selected rounding method is used to convert the input to an integer. If the value of the result is not equal to the input, an inexact exception is raised.

sourceFormat* nextUp *(source0)
The least floating point number that is larger than the input is returned.

sourceFormat* nextDown *(source0)
The largest floating point number that is smaller than the input is returned.

sourceFormat* remainder *(source0, source1)
The modulo operation is performed, that means the remainder of a division has to be computed.

sourceFormat* minNum *(source0, source1)
The smaller floating point number of the two inputs is returned.

sourceFormat* maxNum *(source0, source1)
The larger floating point number of the two inputs is returned.

sourceFormat* minNumMag *(source0, source1)
The smaller absolute value of the two inputs is returned.

sourceFormat* maxNumMag *(source0, source1)
The larger absolute value of the two inputs is returned.

3.2.3.2 LogBFormat operations

sourceFormat* scaleB *(source0, logBFormat)
$x \cdot \beta^n$ is calculated with x being the floating point input, n being an integer input and β being the radix of the floating point number.

logBFormat* logB *(source0)
The logarithm of the floating point input is returned.

3.2.3.3 Arithmetic operations

formatOf* addition *(source0, source1)
The sum of the two floating point inputs is returned.

formatOf* subtraction *(source0, source1)
The difference of the two floating point inputs is returned.

formatOf* multiplication *(source0, source1)
The product of the floating point inputs is returned.

formatOf* division *(source0, source1)
The quotient of the two floating point inputs is returned.

formatOf* squareRoot *(source0)
The square root of the floating point input is returned.

formatOf* fusedMultiplyAdd *(source0, source1, source2)
The sum of the third floating point input and the product of the first and second floating point input without rounding in between is returned.

formatOf* convertFromInt *(int)
The floating point representation of a given integer value is returned.

intFormatOf* convertToIntegerTiesToEven *(source0)
The integer representation of a given floating point number is returned. Halfway cases are rounded to the next even integer number.

intFormatOf* convertToIntegerTowardZero *(source0)
The integer representation of a given floating point number is returned. If the floating point number is not an integral value, the result is rounded towards zero.

intFormatOf **convertToIntegerTowardPositive** ***(source0)***

The integer representation of a given floating point number is returned. If the floating point number is not an integral value, the result is rounded towards $+\infty$.

intFormatOf **convertToIntegerTowardNegative** ***(source0)***

The integer representation of a given floating point number is returned. If the floating point number is not an integral value, the result is rounded towards $-\infty$.

intFormatOf **convertToIntegerTiesToAway** ***(source0)***

The integer representation of a given floating point number is returned. Halfway cases are rounded away from zero.

intFormatOf **convertToIntegerExactTiesToEven** ***(source0)***

The operation $convertToIntegerToEven$ is used and the signaling bit is set to 1 if the result is not equal to the input.

intFormatOf **convertToIntegerExactTowardZero** ***(source0)***

The operation $convertToIntegerTowardZero$ is used and the signaling bit is set to 1 if the result is not equal to the input.

intFormatOf **convertToIntegerExactTowardPositive** ***(source0)***

The operation $convertToIntegerTowardPositive$ is used and the signaling bit is set to 1 if the result is not equal to the input.

intFormatOf **convertToIntegerExactTowardNegative** ***(source0)***

The operation $convertToIntegerTowardNegative$ is used and the signaling bit is set to 1 if the result is not equal to the input.

intFormatOf **convertToIntegerExactTiesToAway** ***(source0)***

The operation $convertToIntegerToAway$ is used and the signaling bit is set to 1 if the result is not equal to the input.

3.2.3.4 Conversion operations

formatOf **convertFromHexCharacter** ***(hexCharacterSequence)***

Converts a hexadecimal representation of a floating point number into the a numerical representation.

hexCharacterSequence **convertToHexCharacter** ***(source0, conversionSpecification)***

Converts a floating point number into a hexadecimal representation.

3.2.3.5 Sign bit operations

sourceFormat **copy** ***(source0)***

The input is returned.

sourceFormat **negate** ***(source0)***

The negated input is returned.

sourceFormat* abs *(source0)
The absolute value of the input is returned.

sourceFormat* copySign *(source0, source1)
The value of the first input with the sign of the second input is returned.

3.2.3.6 Comparision operations

boolean* compareQuietEqual *(source0, source1)
The two floating point inputs are compared. $True$ is returned, iff their values are equal to each other.

boolean* compareQuietNotEqual *(source0, source1)
The two floating point inputs are compared. $True$ is returned, iff their values are not equal to each other.

boolean* compareSignalingEqual *(source0, source1)
If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietEqual* is used otherwise.

boolean* compareSignalingGreater *(source0, source1)
If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietGreater* is used otherwise.

boolean* compareSignalingGreaterEqual *(source0, source1)
If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietGreaterEqual* is used otherwise.

boolean* compareSignalingLess *(source0, source1)
If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietLess* is used otherwise.

boolean* compareSignalingLessEqual *(source0, source1)
If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietLessEqual* is used otherwise.

boolean* compareSignalingNotEqual *(source0, source1)
If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietNotEqual* is used otherwise.

boolean* compareSignalingNotGreater *(source0, source1)
If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietNotGreater* is used otherwise.

boolean* compareSignalingLessUnordered *(source0, source1)
If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietLessUnordered* is used otherwise.

boolean* compareSignalingNotLess *(source0, source1)

If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietNotLess* is used otherwise.

boolean* compareSignalingGreaterUnordered *(source0, source1)

If one of the inputs is a NaN, the signaling bit is set to 1. The operation *compareQuietGreaterUnordere* is used otherwise.

boolean* compareQuietGreater *(source0, source1)

The two floating point inputs are compared. $True$ is returned, iff the value of the first input is greater than the value of the second one.

boolean* compareQuietGreaterEqual *(source0, source1)

The two floating point inputs are compared. $True$ is returned, iff the value of the first input is greater than or equal to the value of the second one.

boolean* compareQuietLess *(source0, source1)

The two floating point inputs are compared. $True$ is returned, iff the value of the first input is less than the value of the second one.

boolean* compareQuietLessEqual *(source0, source1)

The two floating point inputs are compared. $True$ is returned, iff the value of the first input is less than or equal to the value of the second one.

boolean* compareQuietUnordered *(source0, source1)

$True$ is returned iff at least one of the inputs is a NaN, $False$ is returned otherwise.

boolean* compareQuietNotGreater *(source0, source1)

The two floating point inputs are compared. $True$ is returned, iff the value of the first input is not greater than the value of the second one.

boolean* compareQuietLessUnordered *(source0, source1)

$True$ is returned iff at least one of the inputs is a NaN, *compareQuietLess* is used otherwise.

boolean* compareQuietNotLess *(source0, source1)

The two floating point inputs are compared. $True$ is returned, iff the value of the first input is not less than the value of the second one.

boolean* compareQuietGreaterUnordered *(source0, source1)

$True$ is returned iff at least one of the inputs is a NaN, *compareQuietGreater* is used otherwise.

boolean* compareQuietOrdered *(source0, source1)

$True$ is returned iff none of the inputs is a NaN, $False$ is returned otherwise.

3.2.3.7 Conformance predicates

boolean **is754version1985** ***(void)***

$True$ is returned iff the implementation is conform with the IEEE Standard 754-1985 [10].

boolean **is754version2008** ***(void)***

$True$ is returned iff the implementation is conform with the IEEE Standard 754-2008 [11].

3.2.3.8 General non-computational operations

enum **class** ***(source0)***

The category to which the input belongs, is returned. The categories are: $signalingNaN$, $quietNaN$, $negativeInfinity$, $negativeNormal$, $negativeSubnormal$, $negativeZero$, $positiveZero$, $positiveSubnormal$, $positiveNormal$ and $positiveInfinity$.

boolean **isSignMinus** ***(source0)***

$True$ is returned iff the sign bit of the floating point input is set.

boolean **isNormal** ***(source0)***

$True$ is returned iff the floating point input is normal.

boolean **isFinite** ***(source0)***

$True$ is returned iff the floating point input represents a finite value.

boolean **isZero** ***(source0)***

$True$ is returned iff the floating point input is equal to zero.

boolean **isSubnormal** ***(source0)***

$True$ is returned iff the floating point input is subnormal.

boolean **isInfinite** ***(source0)***

$True$ is returned iff the floating point input is equal to $\pm\infty$.

boolean **isNaN** ***(source0)***

$True$ is returned iff the floating point input is NaN.

boolean **isSignaling** ***(source0)***

$True$ is returned iff the signaling bit is set.

boolean **isCanonical** ***(source0)***

$True$ is returned iff the input is canonical.

enum **radix** ***(source0)***

The radix of the input is returned.

boolean **totalOrder** ***(source0, source1)***

$True$ is returned iff the two floating point inputs are totally ordered.

boolean **totalOrderMag** ***(source0, source1)***

$True$ is returned iff the absolute values of the floating point inputs are totally ordered.

3.2.3.9 Flag operations

void **lowerFlags** ***(exceptionGroup)***

The flags specified in the input are set to zero.

void **raiseFlags** ***(exceptionGroup)***

The flags specified in the input are set to 1.

boolean **testFlags** ***(exceptionGroup)***

$True$ is returned iff at least one of the flags specified in the input is set. $False$ is returned otherwise.

boolean **testSavedFlags** ***(flags, exceptionGroup)***

$True$ is returned iff at least one of the flags stored in the input $flags$ specified in the input $exceptionGroup$ is set. $False$ is returned otherwise.

void **restoreFlags** ***(flags, exceptionGroup)***

The flags specified in the input $exceptionGroup$ are overwritten by the states stored in the input $flags$.

flags **saveAllFlags** ***(void)***

The states of all flags are returned.

4 Definition

4.1 Format and special cases

The floating point numbers used in this thesis are binary numbers with variable lengths of mantissa and exponent. The sign requires one bit, the exponent has the length e and the number of bits used for the mantissa is called m. The bias, which is used to depict negative exponents, is called β. It is calculated as described in Section 3.2: $\beta = 2^{e-1} - 1$.
The flags for signaling the exceptions are regarded to be global boolean variables.
Table 3.1 shows the format of all types of floating point numbers including the special values NaN, $\pm\infty$, ± 0 and subnormal floating point numbers.

4.2 Operations

The IEEE Standard 754-2008 [11] specifies, which operations are mandatory for an IEEE-conform implementation. As a matter of course, this thesis defines an arithmetic that includes these operations which are listed and briefly described in Section 3.2.
In the following, the inputs of the operations are marked with a number as index, e. g. $[s_0, e_0, m_0]$ while the ouputs are marked with the letter "r" as index, e. g. $[s_r, e_r, m_r]$.

4.2.1 General computational operations

4.2.1.1 roundToIntegralTiesToEven, roundToIntegralTiesToAway

The operations *roundToIntegralTiesToEven* and *roundToIntegralTiesToAway* round a given floating point number to the next integral floating point number. Except for the used rounding method, they are defined in the same manner shown in Figure 4.1.
Irrespective of the occurence of special cases, the sign of the output is equal to the sign of the input.
If the input is a NaN, $\pm\infty$ or ± 0, the output is equal to the input. If the input is subnormal, it is rounded to zero. Apart from that, the input is checked to be a halfway case. If $m_0 \wedge (1 << (m - (e_0 - \beta) - 1)) == 1$ does not hold, the input is rounded down by cutting off the post-point bits. Otherwise, if additionally $m_0 \wedge (\underbrace{0...0}_{e_0-\beta+1} \underbrace{1...1}_{m-(e_0-\beta)-1}) == 0$ holds, the new mantissa is calculated by cutting off the post-point bits and adding 1.0 depending on the rounding method, it is normalized afterwards if 1.0 was added. The resulting exponent is equal to the sum of e_0 and a normalization factor e'_r, which is either zero or one. If an overflow occurs, it is handled according to the used rounding method.

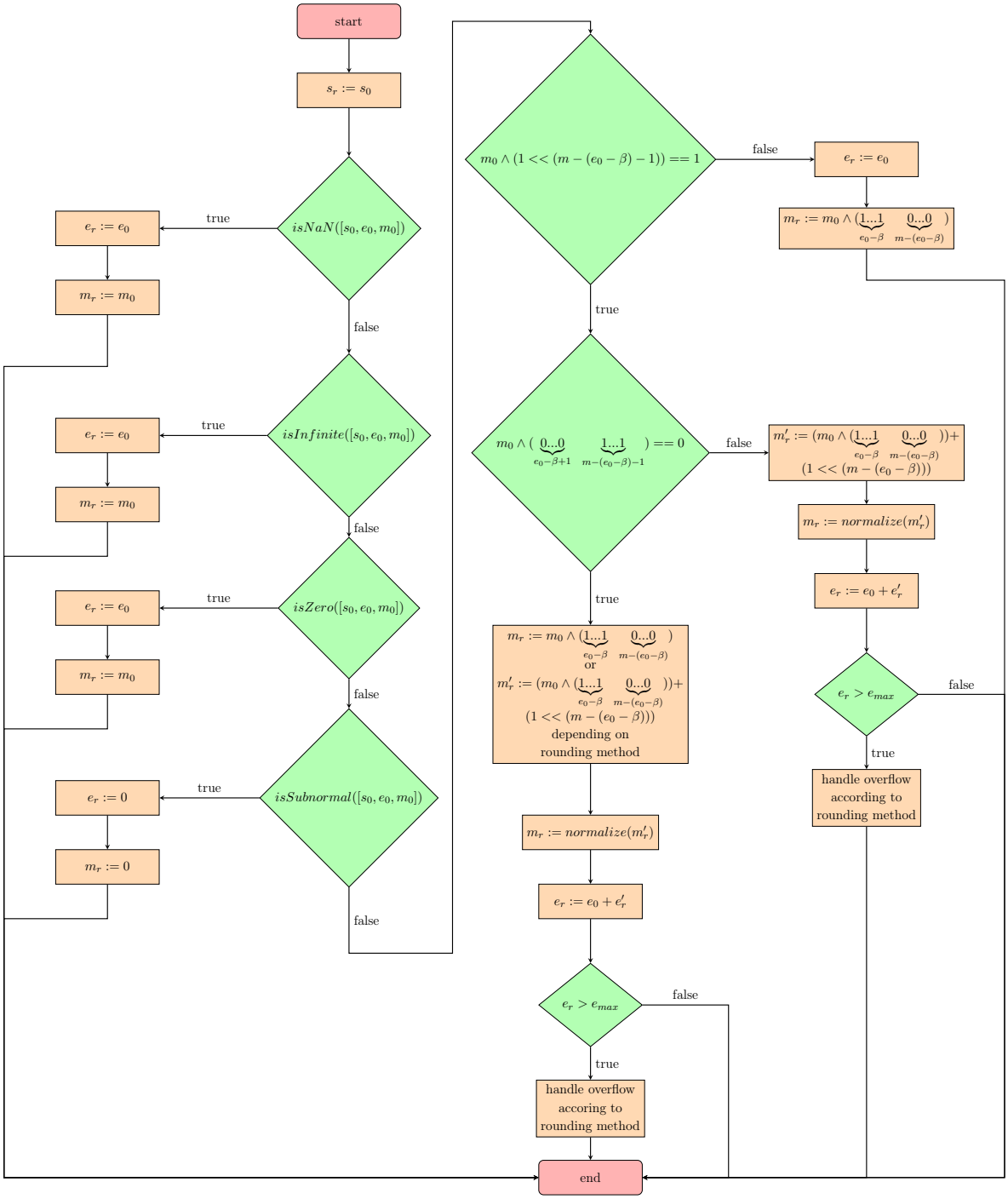

Figure 4.1: Definition of **roundToIntegralTies***

If $m_0 \wedge (\underbrace{0...0}_{e_0-\beta+1} \quad \underbrace{1...1}_{m-(e_0-\beta)-1}) == 0$ does not hold, the resulting mantissa is calculated by cutting off the post-point bits and adding 1.0, afterwards it is normalized. The resulting ex-

ponent is then equal to $e_0 + e'_r$ with e'_r being a normalization factor. If an overflow occurs, it is handled according to the desired rounding method. The offered rounding methods for halfway cases are defined as follows:

roundToIntegralTiesToEven:

$$m'_r = \begin{cases} m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)}) & \text{if } (m_0 \wedge (1 << (e_0 - \beta - 1)) == 1) \\ (m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)})) + (1 << (m - (e_0 - \beta))) & else \end{cases}$$

roundToIntegralTiesToAway:

$$m'_r := (m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)})) + (1 << (m - (e_0 - \beta)))$$

4.2.1.2 roundToIntegralTowardZero, roundToIntegralTowardPositive, roundToIntegralTowardNegative

The operations *roundToIntegralTowardZero*, *roundToIntegralTowardPositive* and *roundToIntegralTowardNegative* round a given floating point number to an integral floating point number. They are all defined in the same manner except for the used rounding method. The shared definition is shown in Figure 4.2.
If the input is a NaN, $\pm\infty$ or ± 0, the output is equal to the input. Apart from that, the input is checked to be an integral value. If $m_0 \wedge (\underbrace{0...0}_{e_0-\beta} \underbrace{1...1}_{m-(e_0-\beta)}) == 0$ holds, the input is already an integral and thus the output is equal to the input. If it does not hold, the mantissa has to be rounded up ($m'_r := m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)}) + (1 << (m - (e_0 - \beta))))$ or down ($m'_r := m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)})$) depending on the used rounding method described. The mantissa has to be normalized afterwards and, corresponding to the normalization, the exponent has to be adjusted. If this adjustment leads to an overflow, it is handled according to the used rounding method.

The offered rounding methods for non-integral inputs are defined as follows:

roundToIntegralTowardZero:

$$m'_r := m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)})$$

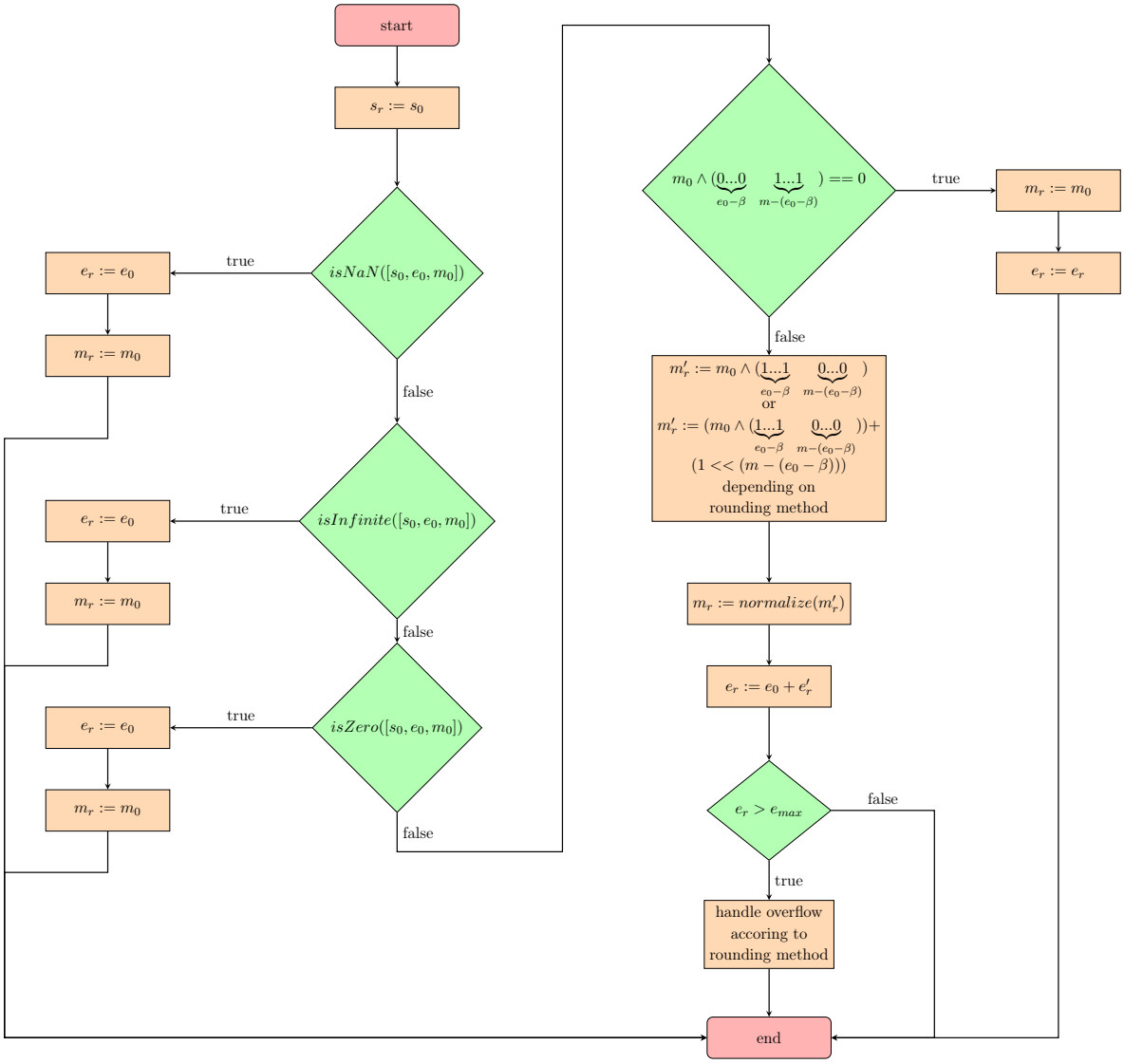

Figure 4.2: Definition of **roundToIntegralToward***

roundToIntegralTowardPositive:

$$m'_r := \begin{cases} (m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)})) + (1 << (m-(e_0-\beta))) & \text{if } (s_0 == 0) \\ m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)}) & else \end{cases}$$

roundToIntegralTowardNegative:

$$m'_r := \begin{cases} (m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)})) + (1 << (m-(e_0-\beta))) & \text{if } (s_0 == 1) \\ m_0 \wedge (\underbrace{1...1}_{e_0-\beta} \quad \underbrace{0...0}_{m-(e_0-\beta)}) & else \end{cases}$$

4.2.1.3 roundToIntegralExact

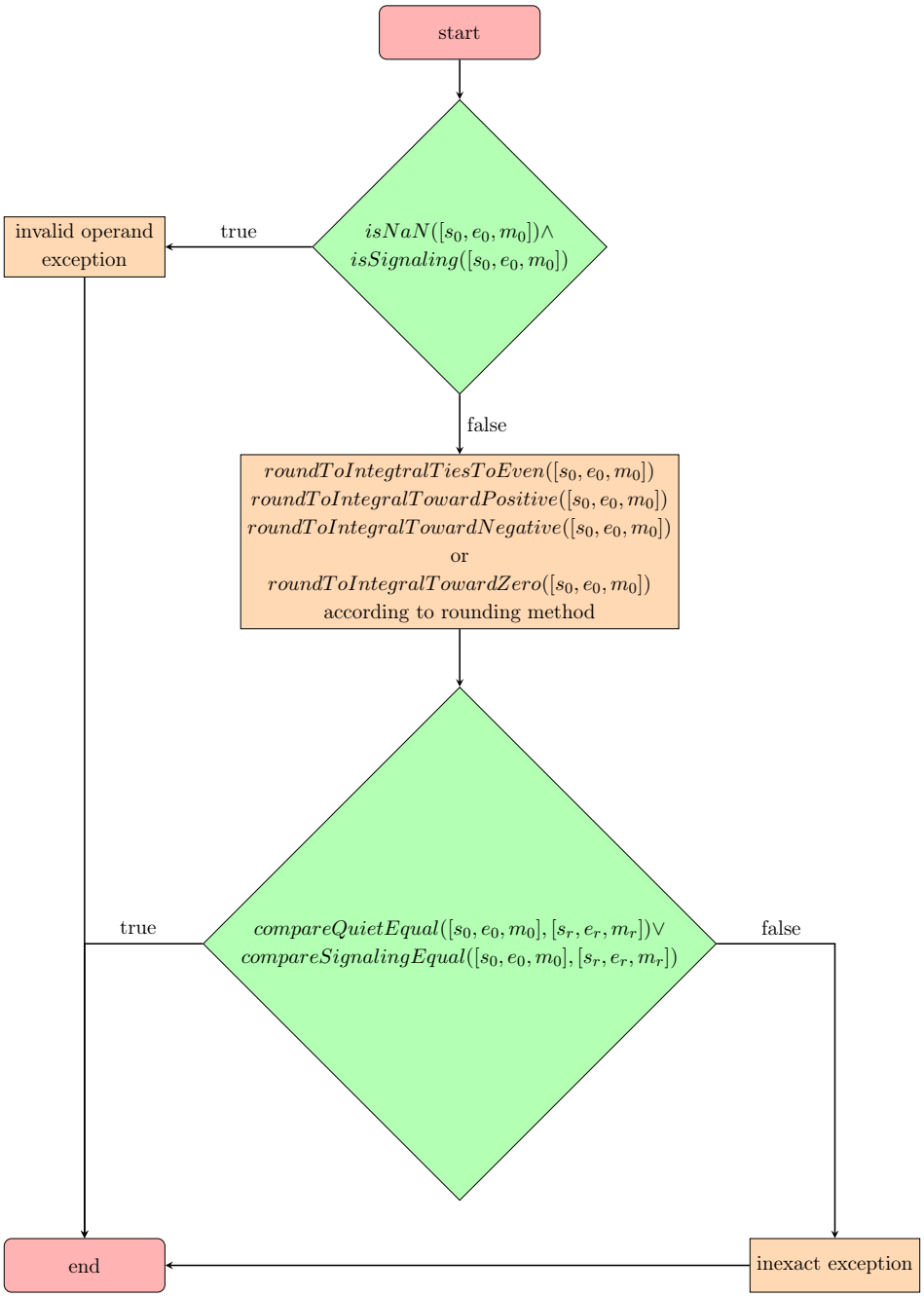

Figure 4.3: Definition of **roundToIntegralExact**

The operation *roundToIntegralExact* defined in Figure 4.3 rounds a given integer to a floating point number representing an integral. If the input is a $sNaN$, the exception that signals an invalid operand has to be thrown. If the input is a number, it is rounded in the next step. The standard rounding method is *roundTiesToEven*, i. e. in most cases,

roundToIntegralTiesToEven is used to calculate the result, but *roundTowardPositive*, *roundTowardNegative* and *roundTowardZero* have to be selectable, too. If *roundTowardPositive* is selected, *roundToIntegralTowardPositive* is used, if *roundTowardNegative* is selected, *roundToIntegralTowardNegative* is used and if *roundTowardZero* is selected, *roundToIntegralTowardZero* is used.

4.2.1.4 nextUp

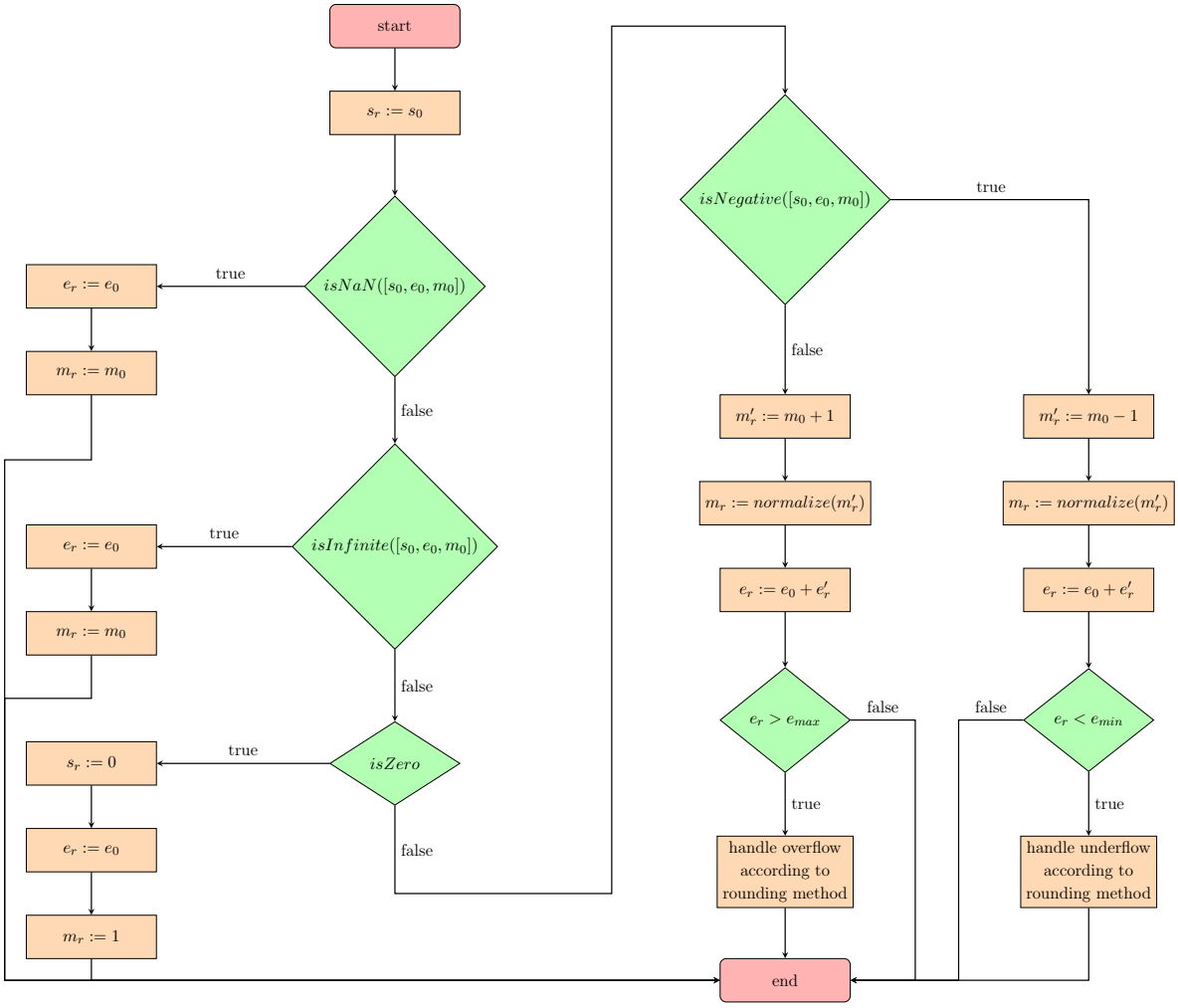

Figure 4.4: Definition of **nextUp**

The operation *nextUp* defined in Figure 4.4 calculates the least floating point value that is larger than the input.
The resulting sign is always equal to the sign of the input, except for one special case.
If the input is equal to ±0, the result is the smallest number, i. e. the resulting exponent is

set to the one of the input, the sign is positive and the mantissa is set to one.
If the input is NaN or $\pm\infty$, the output is equal to the input.
If none of these special cases occur, the sign of the input dictates the adjustment of the mantissa: If the input is negative, the mantissa is decreased by one. If the input is positive, the mantissa is increased by one. Afterwards, it has to be normalized and the resulting exponent is calculated by adding the exponent of the input and a normalization factor. If an underflow or overflow, respectively, occurs, it is handled according to the rounding method.

4.2.1.5 nextDown

The operation *nextdown* is defined as $nextDown([s_0, e_0, m_0]) := -nextUp([\neg s_0, e_0, m_0])$ in the IEEE Standard 175-2008 [11] and it is defined in the same way in this thesis.

4.2.1.6 remainder

The operation *remainder* defined in Figure 4.5 calculates the remainder $[s_r, e_r, m_r]$ of the division of two given finite floating point numbers $[s_0, e_0, m_0]$, $[s_1, e_1, m_1]$.
If at least one of the inputs is a NaN, the output is NaN, too. It is also NaN, if the second input is equal to zero. The IEEE Standard dictates that $remainder(x, \infty)$ has to be x for finite x. Thus the definition described in this thesis returns a NaN if the second input is infinite and the first input is not finite (that is not tantamount to infinite). Otherwise the first input is forwarded to the output if the second input is infinite.
After ensuring that all special cases are considered, the result is calculated in two steps: According to the IEEE Standard, the remainder r can be calculated using the following formula: $r = x - y \cdot n$ with n being the integer nearest to $\frac{x}{y}$. The IEEE Standard also mentions that the remainder has to be calculated regardless of the rounding-direction and if a halfway case occurs, the remainder is always correct, no matter in which direction it is rounded. Thus, the definition in this thesis uses $roundToIntegralTiesToEven$, because all numbers are rounded to the next integral value, as described in the IEEE Standard and halfway cases are rounded to the next even value, what is not in conflict to the IEEE Standard. The first step of the calculation is $[s_n, e_n, m_n] := roundToIntegralTiesToEven(division([s_0, e_0, m_0], [s_1, e_1, m_1]))$ and the second step is $[s_r, e_r, m_r] := fusedMultiplyAdd([\neg s_1, e_1, m_1], [s_n, e_n, m_n], [s_0, e_0, m_0])$.
The IEEE Standard requires that the sign of the result has to be equal to the sign of the first input if the result is zero. Hence, the result is tested for being zero to assign the correct sign accordingly.

4.2.1.7 minNum

The operation $minNum$ defined in Figure 4.6 returns the smaller one of two given floating point numbers. The result is given as $[s_r, e_r, m_r]$ while the two inputs are given as $[s_0, e_0, m_0]$ called $input_0$ and $[s_1, e_1, m_1]$ called $input_1$.
First of all the inputs are checked for being $sNaN$. If $input_0$ is $sNaN$, it is returned. Otherwise, $input_1$ is checked for being $sNaN$. Afterwards, both inputs are tested for being $qNaN$. If $input_0$ is applied to be $qNaN$, $input_1$ is returned and vice versa according to Section 4.1.

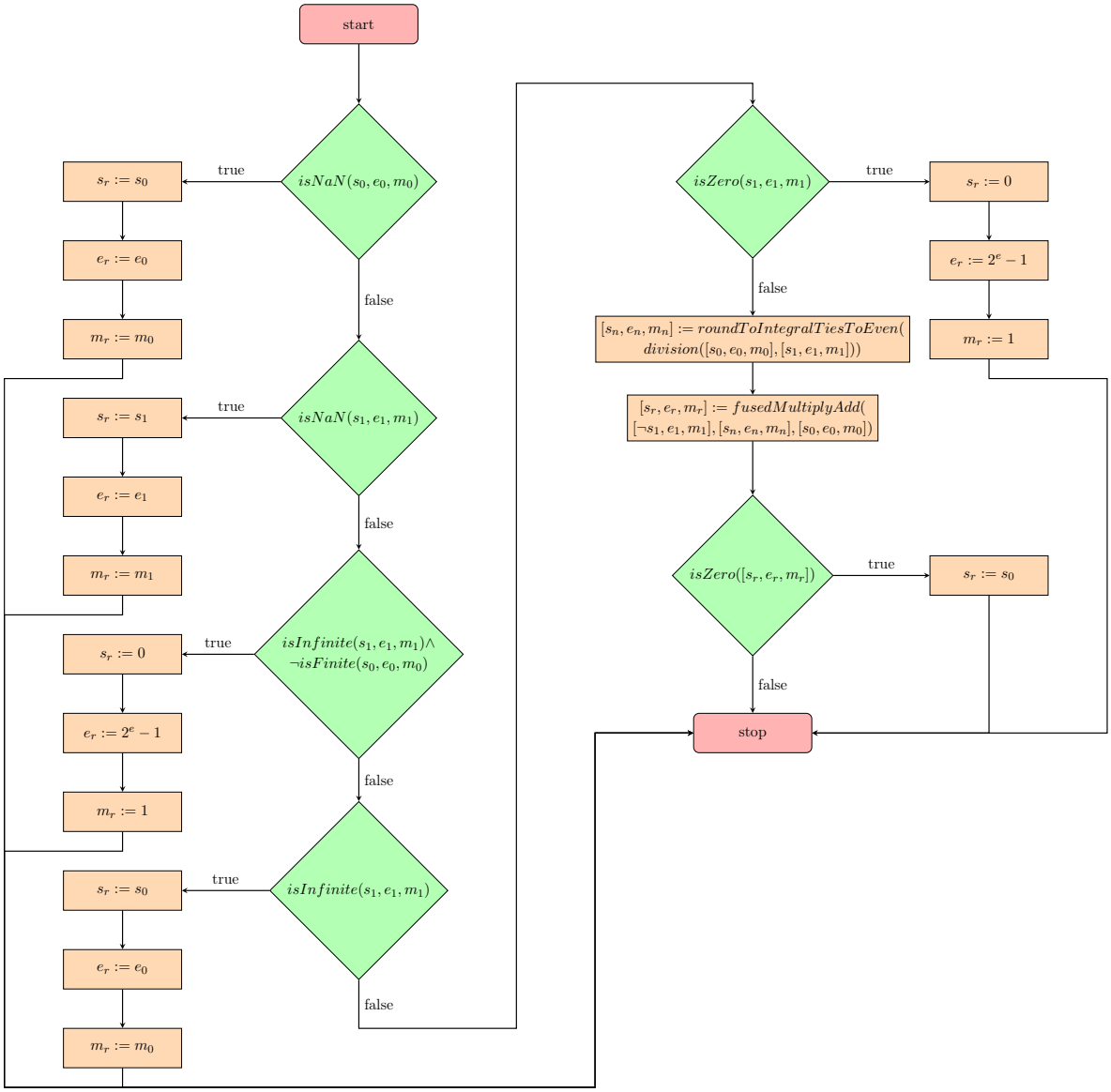

Figure 4.5: Definition of **remainder**

The operations *isNaN* and *isSignaling* defined below are used to check for special cases. The operations *compareQuietLess* and *compareSignalingLess* are subsequently used to determine, which of the two inputs is the minor one.

4.2.1.8 maxNum

The operation *maxNum* defined in Figure 4.7 returns the larger one of two given floating point numbers. The result is given as $[s_r, e_r, m_r]$ while the two inputs are given as $[s_0, e_0, m_0]$ called $input_0$ and $[s_1, e_1, m_1]$ called $input_1$.
Special cases of the operation *maxNum* defined in Figure 4.7 are handled in the same manner

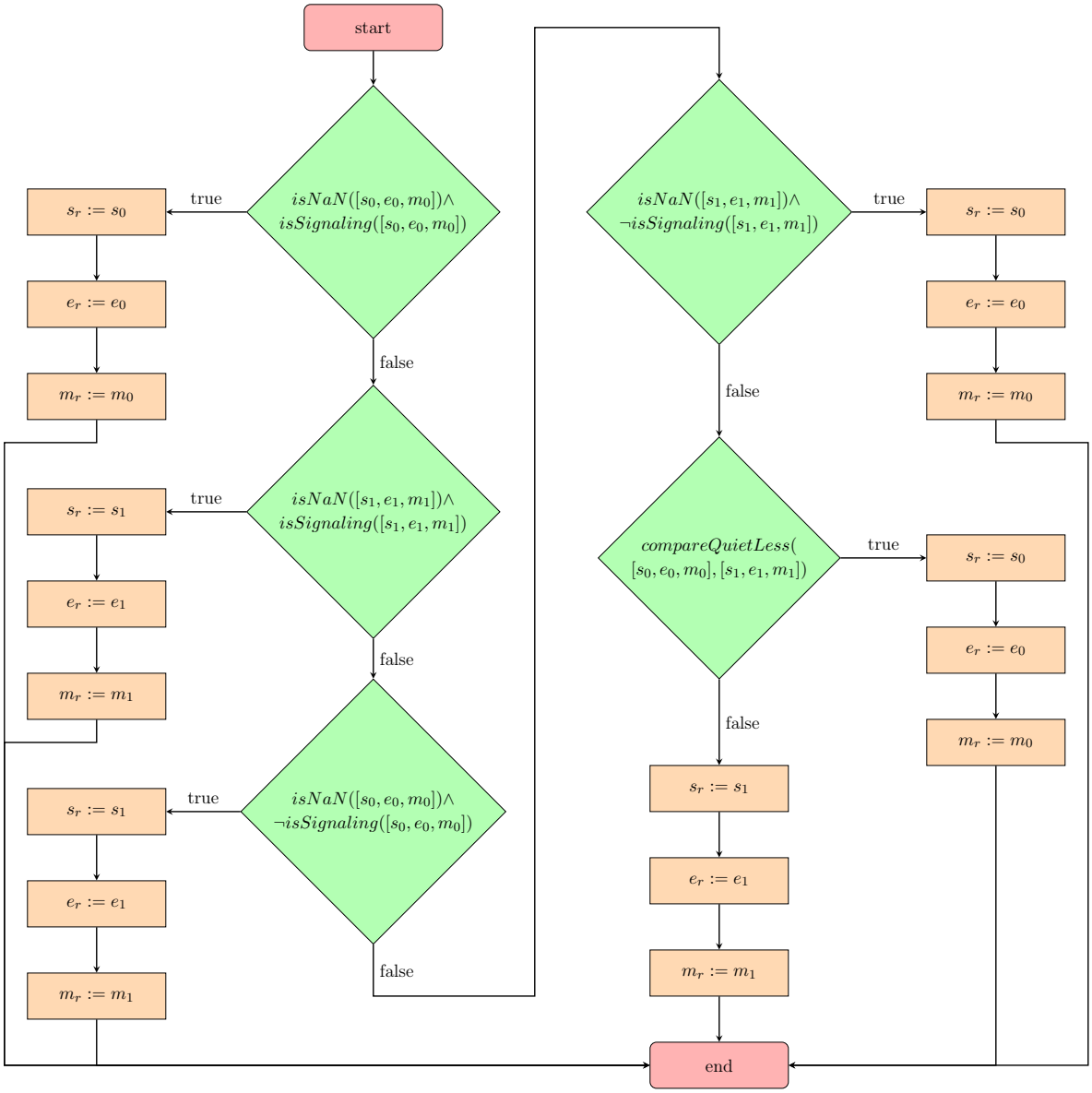

Figure 4.6: Definition of **minNum**

as in the operation $minNum$ described above. After eliminating the occurence of special cases, the operations $compareQuietGreater$ and $compareSignalingGreater$ are used to identify the major input.

4.2.1.9 minNumMag

The difference of the operation $minNumMag$ defined in Figure 4.8 and the operation $minNum$ described above is that the absolute values are compared after all special cases are ruled out. This is done by applying the operation abs defined in Figure 4.26 to the inputs prior to the

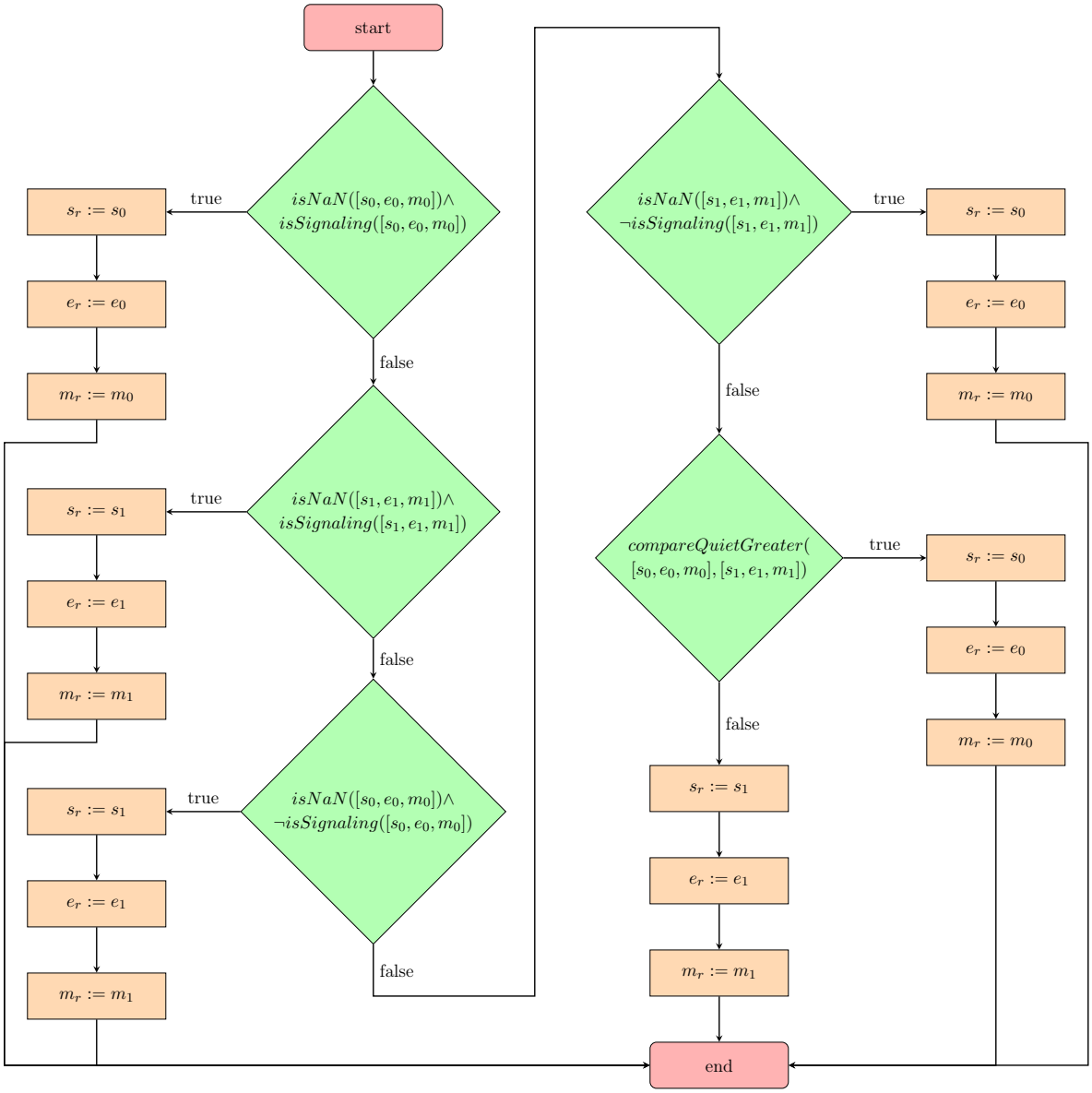

Figure 4.7: Definition of **maxNum**

comparison realized by using *compareQuietLess* and *compareSignalingLess*.
Although the change of the sign would not change the meaning of $sNaN$ or $qNaN$, simply replacing the operation $minNumMag$ by applying *abs* to both inputs followed by the operation $minNum$ with the modified inputs would not always lead to desired result. As described in Section 3.2, the sign of resulting NaNs depends on the inputs. But this information would be erased by applying *abs* before looking for special cases. Therefore, the $minNum$ is not used to realize $minNumMag$.

Figure 4.8: Definition of **minNumMag**

4.2.1.10 maxNumMag

The definition of *maxNumMag* depicted in Figure 4.9 exhibits a similar appearance to the definition of *minNumMag* shown in Figure 4.8. The only difference is that the result is the major input iff no special cases occur. This is done by using *compareQuietGreater* and *compareSignalingGreater* after applying *abs* to the inputs.

Figure 4.9: Definition of **maxNumMag**

4.2.2 LogBFormat operations

4.2.2.1 scaleB

The operation *scaleB* defined in Figure 4.10 calculates $x \cdot \beta^n$ with $x = [s_0, e_0, m_0]$ given as floating point number, n given as integer and β is the radix of the floating point number, what is 2 for this arithmetic.
The output is equal to the input x, if x is a NaN or equal to $\pm\infty$ or ± 0 or if n is equal to zero.

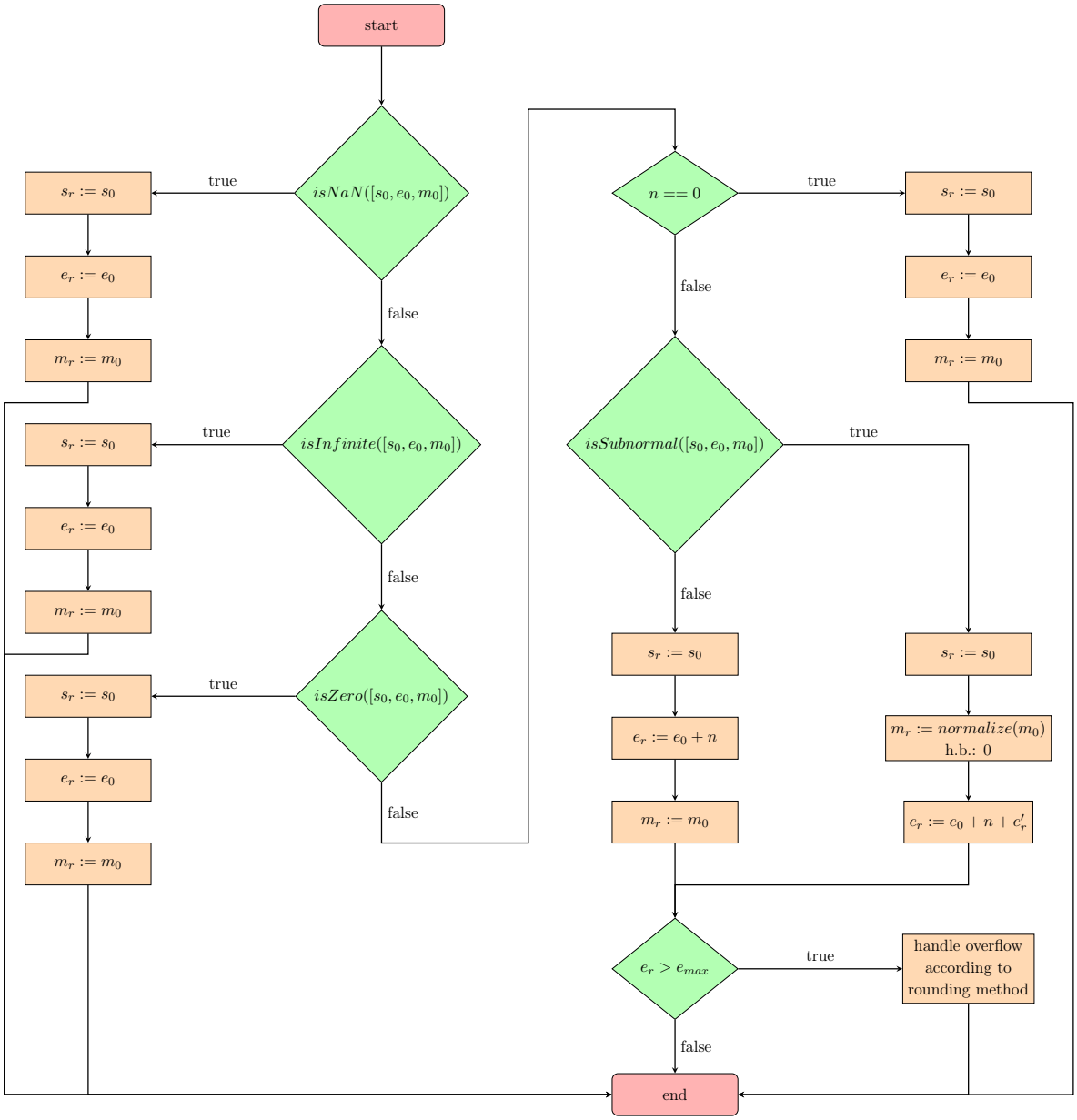

Figure 4.10: Definition of **scaleB**

If none of these special cases occur, the exponent of the result is calculated as $(-1)^{s_0} \cdot 2^{e_0+n} \cdot m_0$ considering the hidden bit: If x is subnormal, the result has to be normalized.
After accomplishing the calculation, the result is checked for the occurence of an overflow what is then handled according to the rounding method.

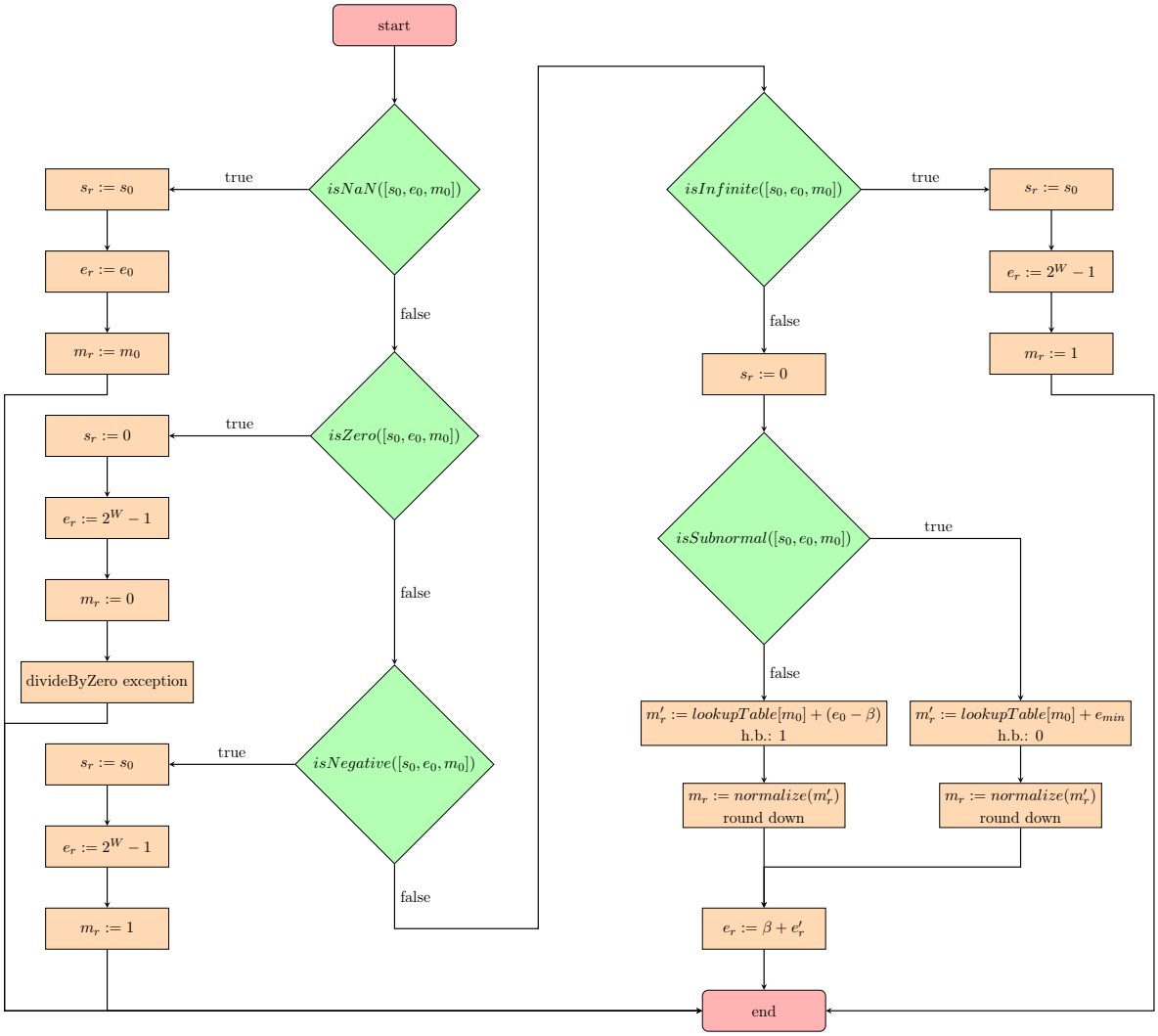

Figure 4.11: Definition of **logB**

4.2.2.2 logB

The operation $logB$ calculates the logarithm of a given floating point number.
Vinyals et al. invented an logarithm implementation and described it in [26] what is based on a lookup table containing log_2 of all possible mantissas. The arithmetic defined in this thesis makes use of that implementation.
If the input is NaN, the ouput is equal to the input. The output is also NaN, if the input is negative but unequal to zero. The remaining special cases are $log_2(\pm 0) := -\infty$, what also raises a divideByZero exception and $log_2(+\infty) := +\infty$.
If none of these special cases apply, the calculation is done as described by Vinyals et al. in [26]. That calculation is based on the equation $log_2(2^{e_0-\beta} \cdot 1.m_0) = (e_0 - \beta) + log_2(1.m_0)$ for normal and $log_2(2^{e_{min}} \cdot 0.m_0) = e_{min} + log_2(0.m_0)$ for subnormal floating point numbers. So

it is sufficient to store the logarithms for all mantissas in a lookup table to be able to calculate the logarithm of a floating point number. The logarithm is only defined for positive numbers, so it is know, that the result is positive. The binary logarithm of the mantissa is looked up from the lookup tables mentioned above and added to the inputs exponent to calculate the mantissa of the result. The mantissa may have to be normalized by using round down as rounding method. The exponent of the result is equal to the sum of β and a normalization factor. An overflow cannot occur, because the normalization factor is either zero or one.

4.2.3 Arithmetic operations

4.2.3.1 addition

The operation *addition* depicted in Figure 4.12 computes the sum $[s_r, e_r, m_r]$ of two given floating point numbers $[s_0, e_0, m_0]$ and $[s_1, e_1, m_1]$.
Before the actual computation of the sum can be started, the occurence of special cases has to be ruled out, that means all combinations of inputs that lead to a NaN as result. $+\infty+(-\infty)$ as well as $-\infty+(+\infty)$ cannot be evaluated non-ambiguous, so these cases result in a NaN, in this thesis with a positive sign because it is not defined in the IEEE Standard 754-2008. Another problem is the calculation with at least one input that is NaN, which also leads to a NaN as output. Even if that is contradictory with commutativity, the sign is taken from the first NaN according to the IEEE Standard 754-2008 [11].
The commutativity permits the next step: To ensure that the absolute value of the first summand is greater or equal to the absolute value of the second summand, the inputs are swapped if $(e_0 < e_1) \vee ((e_0 == e_1) \wedge (m_0 < m_1))$ holds. The next prearrangement for the addition is converting the mantissa of the second summand into a 2-complement number considering its hidden bit. If the signs of both inputs are different, a subtraction instead of an addition has to be performed. That can be realized by negating the second summand. The next step is shifting the second summand right using shift arithmetic right (SAR) and adjusting its exponent accordingly until both exponents are equal to each other. During this shifting, the bits G (guard bit), R (round bit) and S (sticky bit) are identified. G is the last bit that is cut off during the shifting, R is the second last one and S is set to 1 iff at least one of the remaining cut off bits is not zero.
After completing all these prearrangements, the addition is performed as a 2-complement addition considering the hidden bit of the first summand. Afterwards, the sum is normalize, the resulting exponent is calculated and the sign is determined. The normalization procedure is depicted in Figure 4.13. If an overflow occured during the addition, the mantissa has to be shifted to the right. The rounding bits G, R and S have to be adjusted, G becomes the least bit of the mantissa, R becomes the old G and S becomes the disjunction of the old S and R because it indicates if at least of the bits right of R is non-zero. The resulting exponent has to be increased by one. If no overflow occured, the mantissa has to be shifted to the left if the result is not subnormal and the mantissa is less than 1.0. The resulting exponent is decreased by one, G becomes the old R, R becomes false, S keeps its value and the mantissa is shifted to the left while G becomes the least bit of the mantissa. That has to be repeated until the

mantissa is normalized. Regardless of the adjustments before, the mantissa has to be rounded subsequently according to the rounding method with respect to G, R and S.
The defined method is also described by Schürmann in [24] and Muller et al. in [18].

4.2.3.2 subtraction

A subtraction can easily be done by changing the sign of the subtrahend and perform an addition. This is done by the *subtraction* shown in Figure 4.14. The sole exception is the case that the subtrahend is NaN. If that is the case, the sign of the result is equal to the sign of the subtrahend and if the addition is used to calculate the result, the subtrahend is negated. The minuend is tested to be NaN, too, because the if both inputs are NaN, the sign of the minuend has to be used as sign of the result. And that would not be the case if only the subtrahend is tested to be NaN.
After excluding NaN being one of the inputs, the sign of the subtrahend is negated and the addition defined in Figure 4.12 is used to calculate the result.

4.2.3.3 multiplication

The operation *multiplication* defined in Figure 4.15 computes the product $[s_r, e_r, m_r]$ of two given inputs $[s_0, e_0, m_0]$ and $[s_1, e_1, m_1]$.
First, the occurence of special cases is checked. $0 \cdot \infty$ as well as $\infty \cdot 0$ results in a NaN. The sign of the product "is the exlusive OR of the operands' signs" [11]. The sign of the resulting NaN is calculated the same way. Apart from that, a NaN as result is only possible, iff one of the inputs is a NaN. If so the sign is taken from the first input that is NaN regarding to [11]. All successive cases do not result in a NaN, so the sign is always $s_r := s_0 \oplus s_1$ from now on as stipulated in [11]. If both multiplicands are ∞, the result is also ∞ and analogous, if both multiplicands are 0, the result is also 0.
After excluding the occurence of the described special cases, the actual calculation of the product can be started. Before going into the calculation, the interpretation of the given data has to be clear. The given floating point numbers are checked to be subnormal. That leads to four possible cases:

Case 1: $[s_0, e_0, m_0]$ is normal and $[s_1, e_1, m_1]$ is normal.
$\Rightarrow$ The hidden bit of both, input_0 and input_1, is 1.

Case 2: $[s_0, e_0, m_0]$ is normal and $[s_1, e_1, m_1]$ is subnormal.
$\Rightarrow$ The hidden bit of input_0 is 1, the hidden bit of input_1 is 0.

Case 3: $[s_0, e_0, m_0]$ is subnormal and $[s_1, e_1, m_1]$ is normal.
$\Rightarrow$ The hidden bit of input_0 is 0, the hidden bit of input_1 is 1.

Case 4: $[s_0, e_0, m_0]$ is subnormal and $[s_1, e_1, m_1]$ is subnormal.
$\Rightarrow$ The hidden bit of both, input_0 and input_1, is 0.

Under the designation of the hidden bits, the new mantissa is calculated by multiplying input_0 and input_1. Subsequently the calculated mantissa is normalized, before the new exponent is

calculated by adding the exponents of the multiplicands, adding the bias β and contingently adding an adjusting parameter which is a result of the normalization. Finally, the existence of an overflow or an underflow is checked and, if one occurs, it is handled according to the selected rounding method.

4.2.3.4 division

The operation *division* depicted in Figure 4.16 computes the quotient $[s_r, e_r, m_r]$ of two given floating point numbers, $[s_0, e_0, m_0]$ and $[s_1, e_1, m_1]$.
The handling of resulting $NaNs$ corresponds to the handling during a multiplication, but the calculations that result in a NaN are different. $\frac{0}{0}$ and $\frac{\infty}{\infty}$ lead to a NaN as result. The sign is calculated in the same manner as for normal divisions: $s_r := s_0 \oplus s_1$. If one of the inputs is a NaN, it is handled as described for the operation *multiplication*.
The further procedure is equivalent to the one described above, except for the calculation of the resulting exponent: The exponent of the divisor is subtracted and the bias β is added.

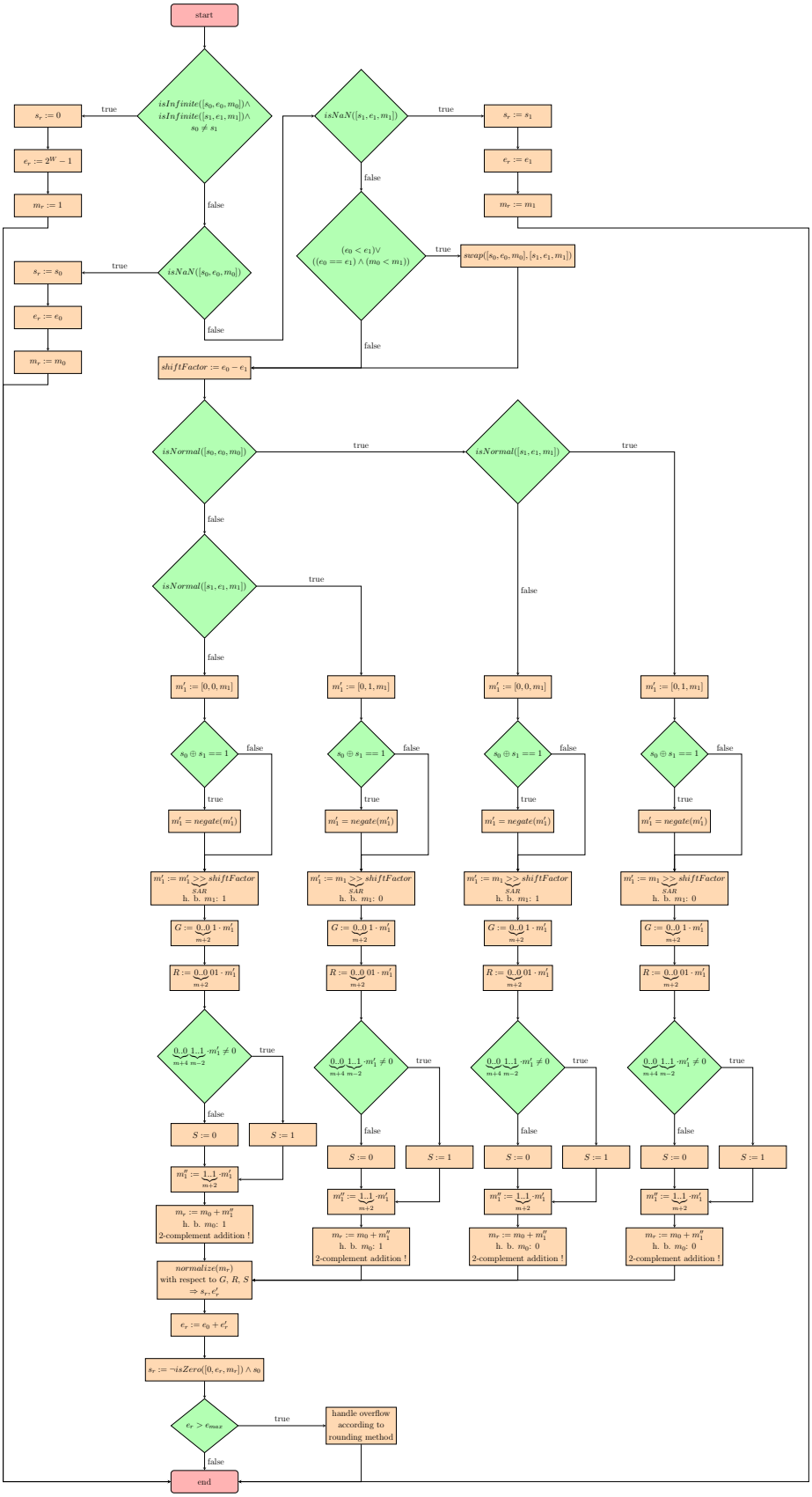

Figure 4.12: Definition of **addition**

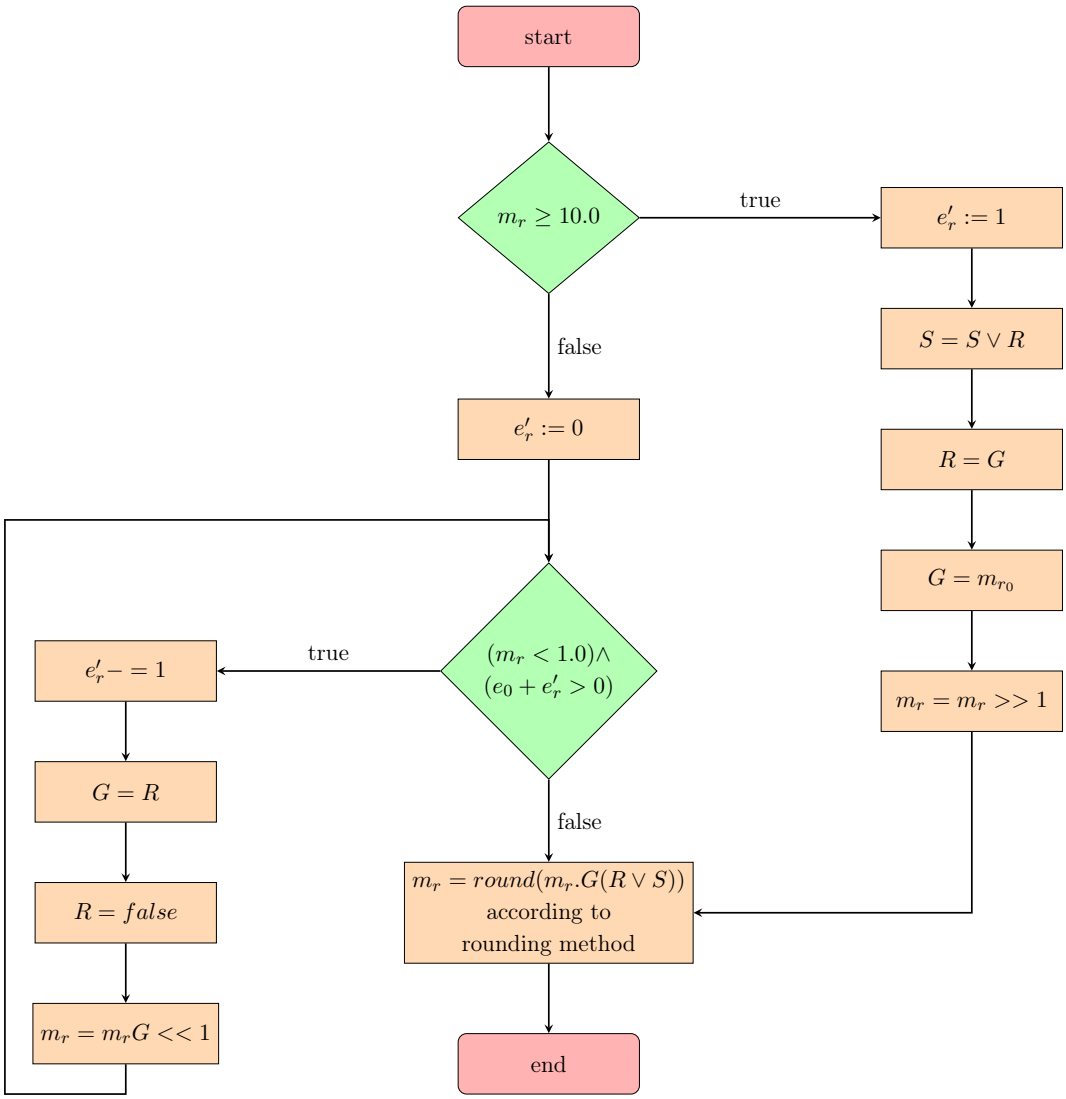

Figure 4.13: Definition of Normalization In **addition**

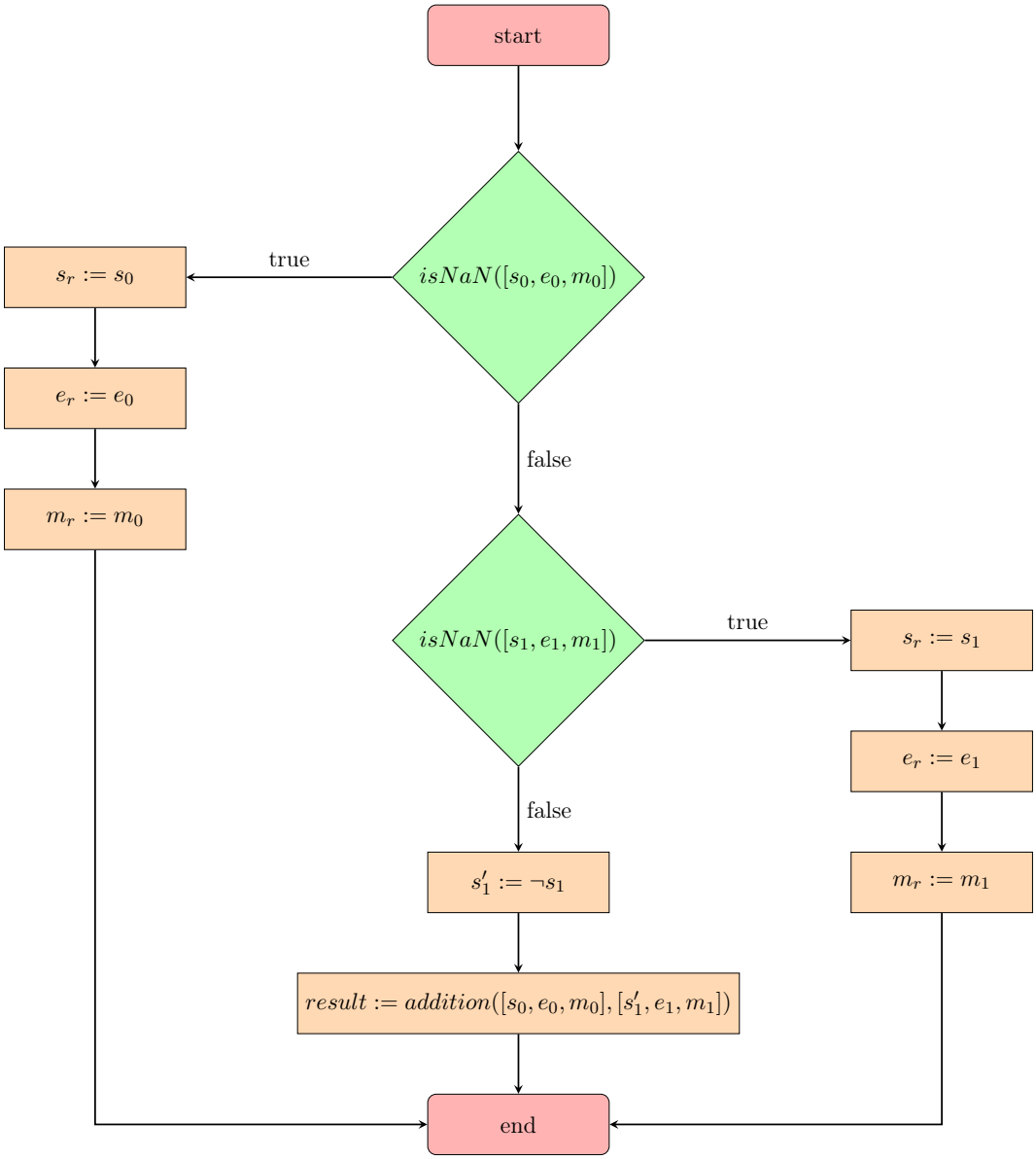

Figure 4.14: Definition of **subtraction**

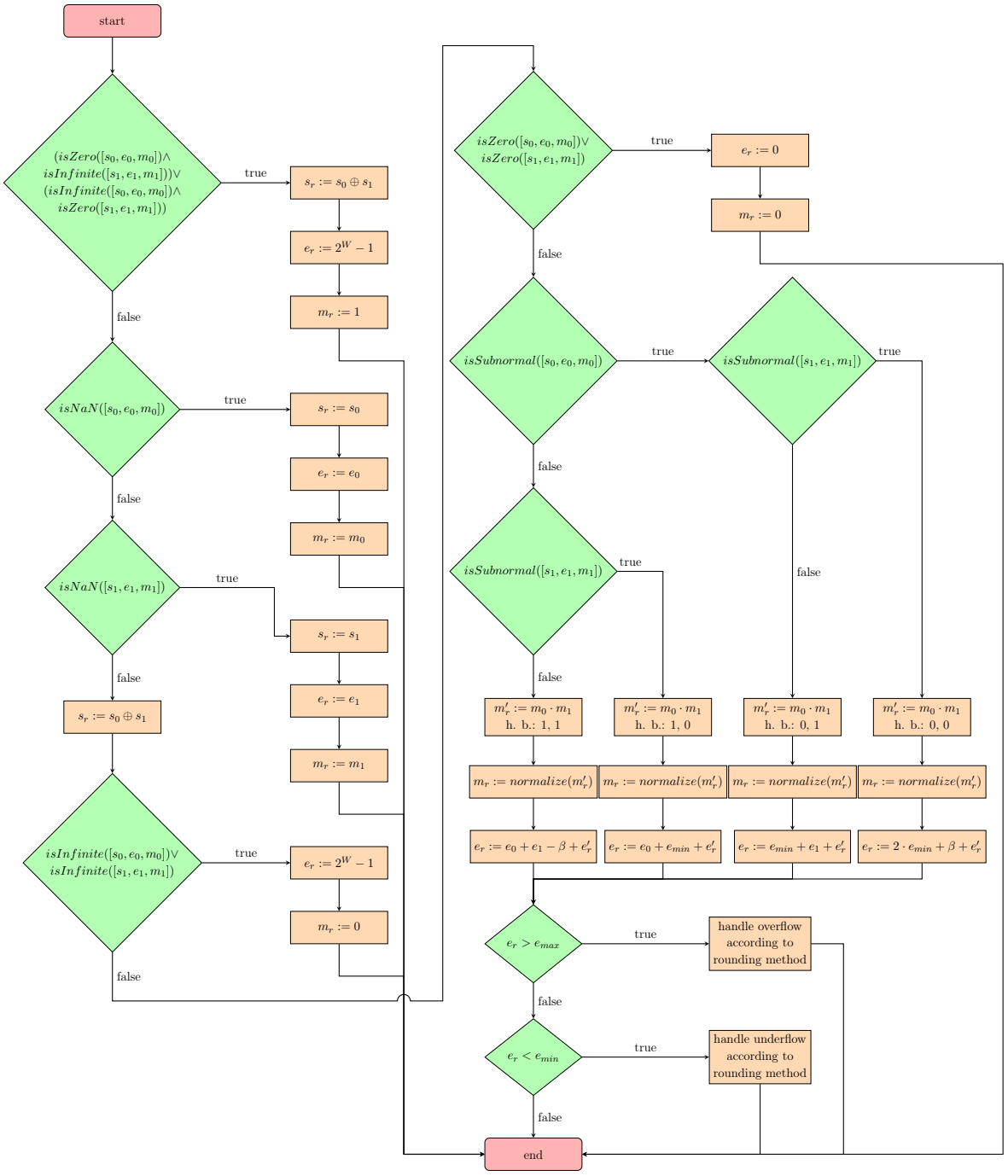

Figure 4.15: Definition of **multiplication**

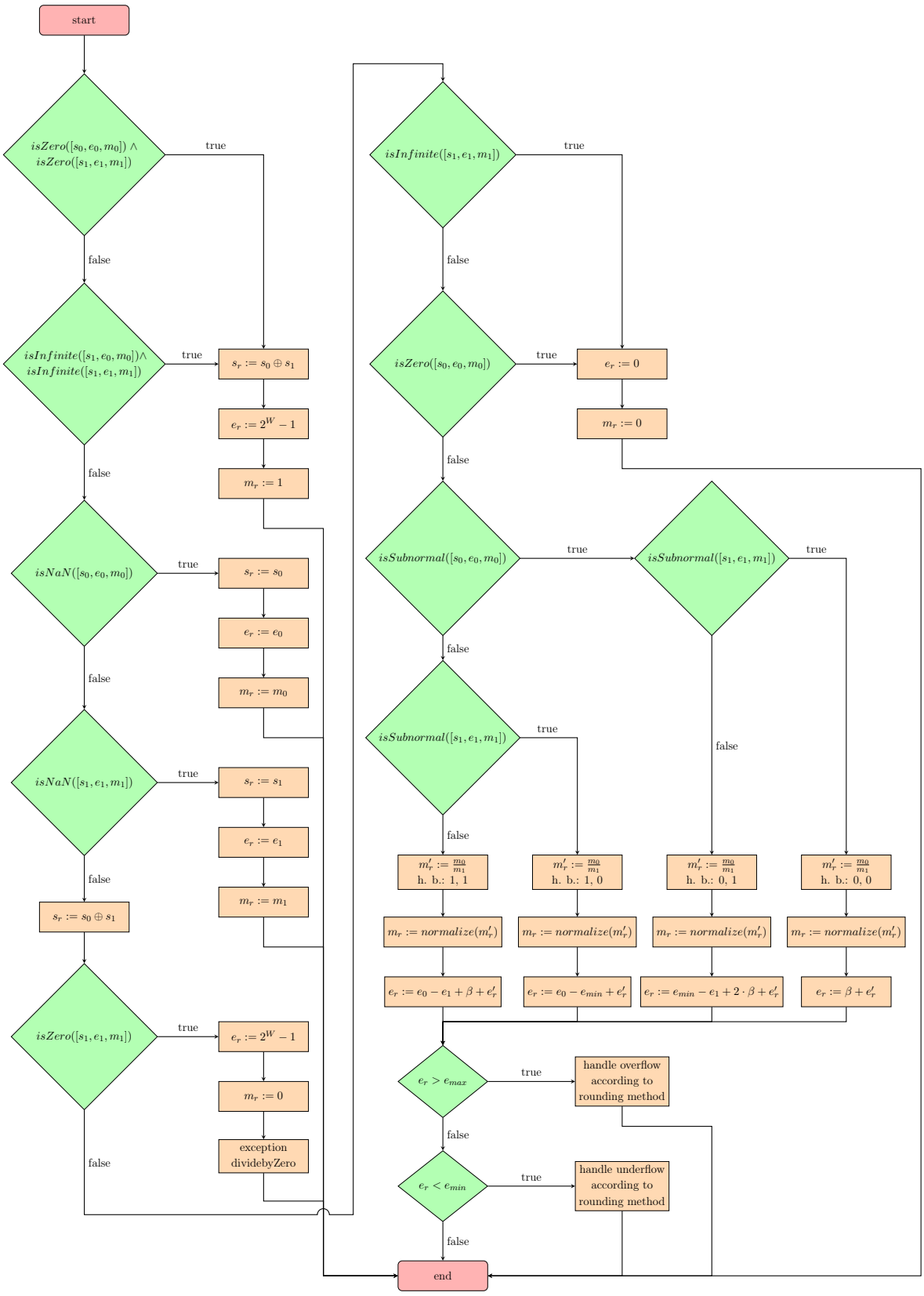

Figure 4.16: Definition of **division**

4.2.3.5 squareRoot

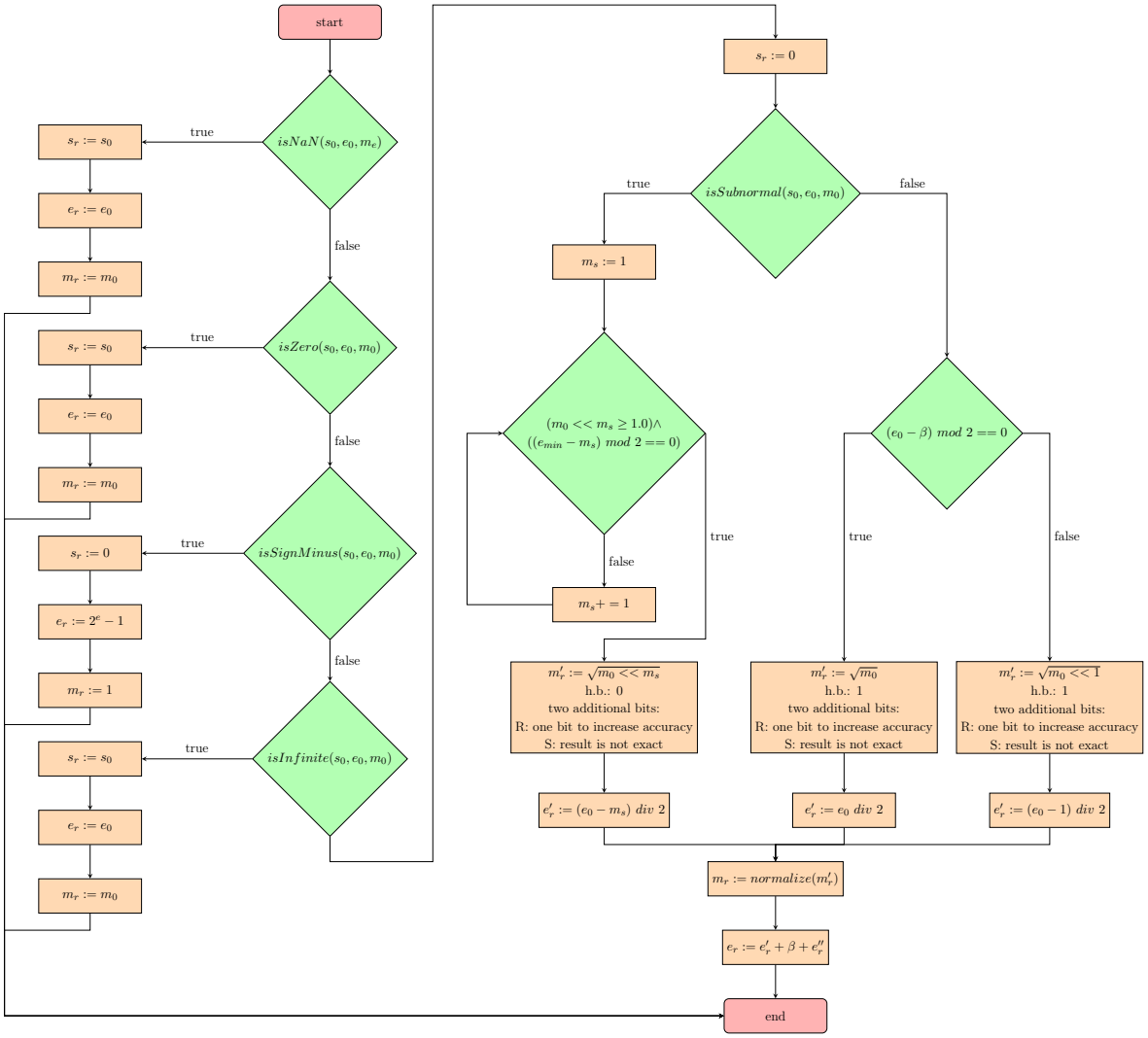

Figure 4.17: Definition of **squareRoot**

The operation *squareRoot* depicted in Figure 4.17 computes the square root $[s_r, e_r, m_r]$ of given floating point number $[s_0, e_0, m_0]$. The special cases are handled according to the definition of Muller et al. in [18]. If the input is an NaN or a negative but unequal to zero, the output is $qNaN$. If the input is equal to zero, the output is zero, too, while the sign of the output is equal to the sign of the input. This case has to be handled separately, because the IEEE Standard 754-2008 defines that $\sqrt{-0} = -0$ has to apply. For an infinite input, the output is also equal to the input.
If none of the described special cases occur, the sign of the result is known to be positive. The square root of a product can be split into the product of two square roots according to [15]: $\sqrt{x \cdot y} = \sqrt{x} \cdot \sqrt{y}$ and the square root can be written as exponent: $\sqrt{x^y} = x^{\frac{y}{2}}$. To ease the

calculation of the square root of the mantissa, it is shifted until it is has exactly one bit on the left hand side of the point. As a matter of course, that can only occur if the input is subnormal, because the hidden bit of normal numbers is 1 according to the definition. If the adapted exponent is odd, the mantissa is shifted one additional bit to the left to ensure an integral exponent.
After arranging mantissa and exponent, the square root of the mantissa is performed. Two additional bits are returned: The round bit (R) and the sticky bit (S) are required to perform the normalization. The square root of the exponent part is calculated by dividing the exponent by 2. Afterwards, the normalization has to be executed. If $R = 1$ holds, S is used to determine the rounding direction according to the chosen rounding method. If $R = 0$ holds, the mantissa can be used without normalization. Finally, the resulting exponent is calculated with respect to the normalization factor.

4.2.3.6 fusedMultiplyAdd

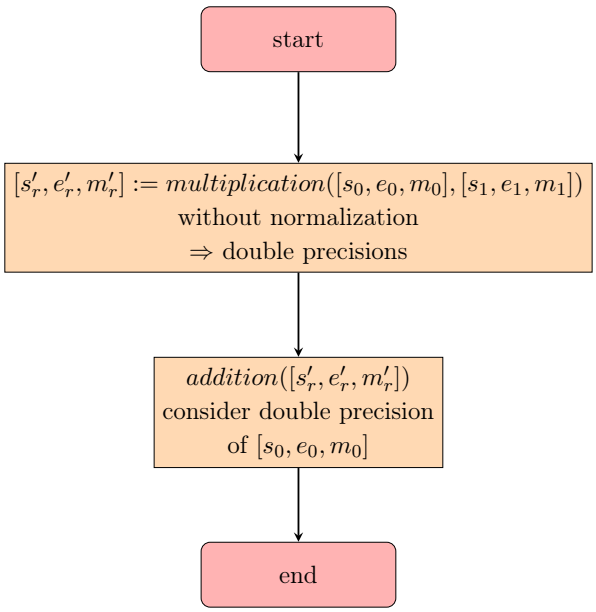

Figure 4.18: Definition of **fusedMultiplyAdd**

The operation $fusedMultiplyAdd$ calculates $input_0 \cdot input_1 + input_2$ without rounding after multiplying. The definition shown in Figure 4.18 uses the definition of a normal mulitplication shown in Figure 4.15 but without normalization of the result. Afterwards, it is added to $input_2$ with usage of the normal addition defined in Figure 4.12 and described above but with respect to the not rounded product calculated before. The result is normalized the n.

4.2.3.7 convertFromInt

The operation $convertFromInt$ defined in Figure 4.19 calculates the floating point representation of a given integer.

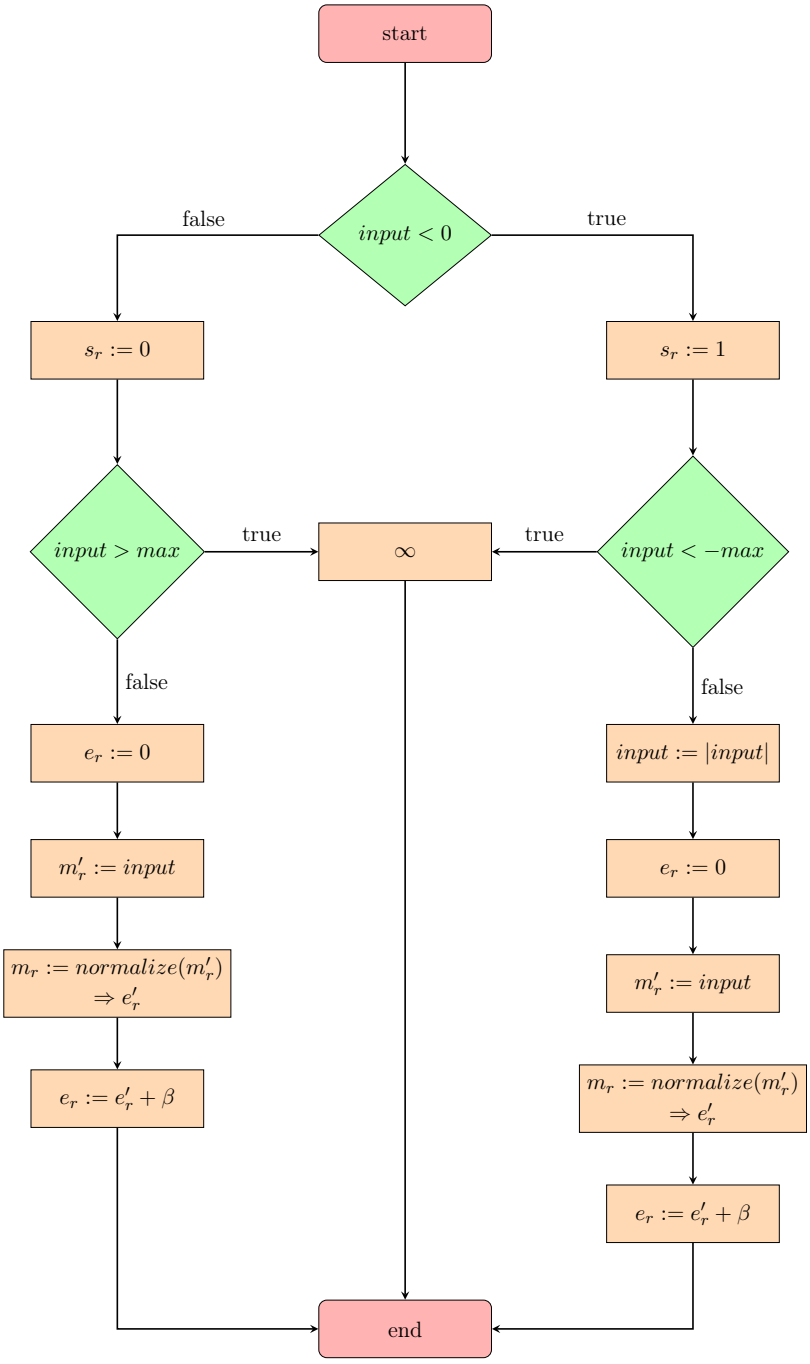

Figure 4.19: Definition of **convertFromInt**

The integer format does not contain special values like NaN or a representation of ∞. So it can be assumed, that the input is a number. First of all, the sign is determined: If the input is a negative number, the sign is 1, otherwise it is 0. An integer can only be a datum of integral numbers, so the following holds: $\forall x \in \mathbb{Z} : (x == \pm 0) \vee (|x| \leq 1)$. Because of that, it is clear, that the only output that is not a representable floating point number, is $\pm\infty$. The output

is ∞ iff the input is either larger than the maximal or less than the minimal representable floating point number. If the number is representable, then the input is considered as positive from now on. The exponent is set to zero while the mantissa gets the value of the input. The mantissa is then normalized and the bias is added to the received exponent.

4.2.3.8 convertToIntegerTiesToEven, convertToIntegerTowardZero, convertToIntegerTowardPositive, convertToIntegerTowardNegative, convertToIntegerTiesToAway, convertToIntegerExactTiesToEven, convertToIntegerExactTowardZero, convertToIntegerExactTowardPositive, convertToIntegerExactTowardNegative, convertToIntegerExactTiesToAway

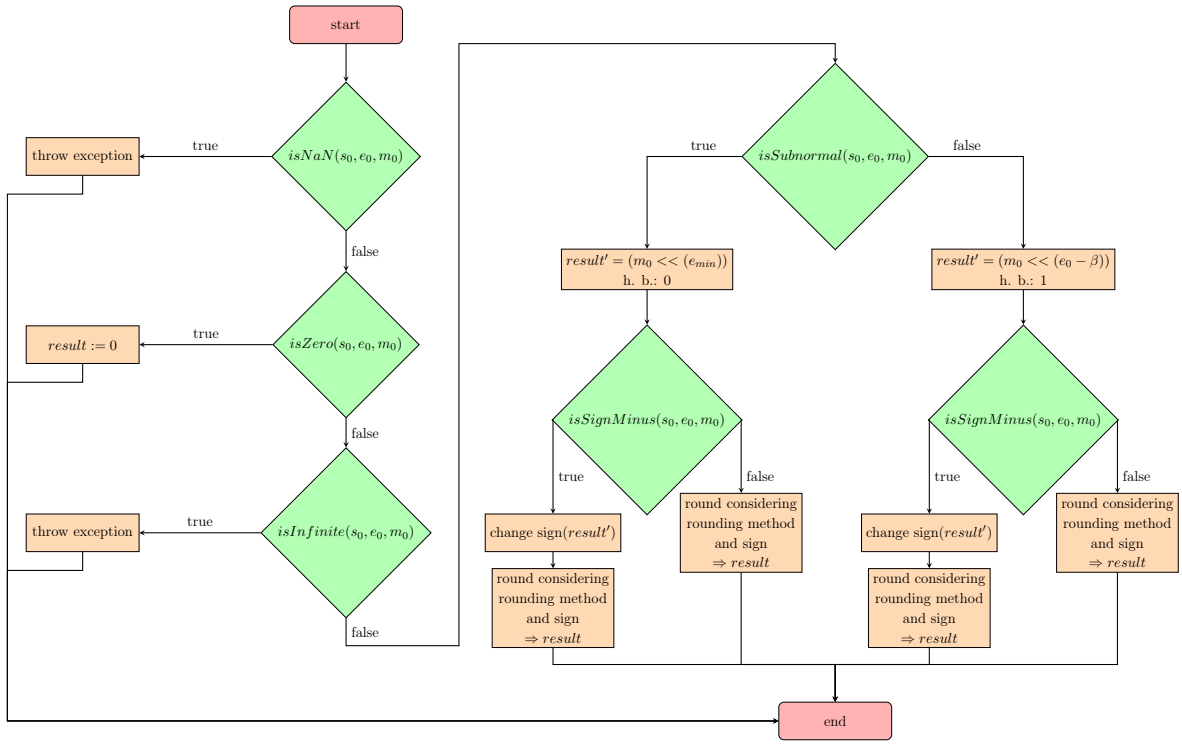

Figure 4.20: Definition of **convertToInteger***

The operations *convertToIntegerTiesToEven*, *convertToIntegerTowardZero*, *convertToIntegerTowardPositive*, *convertToIntegerTowardNegative*, *convertToIntegerTiesToAway*, *convertToIntegerExactTiesToEven*, *convertToIntegerExactTowardZero*, *convertToIntegerExactTowardPositive*, *convertToIntegerExactTowardNegative* and *convertToIntegerExactTiesToAway* con-

vert a given floating point number to an integer. Except the used rounding method, they all are defined in the same manner as shown in Figure 4.20. If the input is either a NaN or equal to $\pm\infty$, an exception has to be thrown. If the input is equal to ± 0, the output is zero, because a negative zero does not exist for integers. If none of these special cases occur, the mantissa is shifted according to the exponent and corresponding to the hidden bit. If the input was a negative floating point number, the extension has to be negated. The final result is calculated by rounding to an integer.
The used rounding methods are defined as follows:

convertToIntegerTiesToEven:

$$roundTiesToEven(x) = \begin{cases} \lfloor x \rfloor & \text{if } (x - \lfloor x \rfloor < 0.5) \\ \lceil x \rceil & \text{if } (x - \lfloor x \rfloor > 0.5) \\ \lfloor x \rfloor & \text{if } (\lfloor x \rfloor \ mod\ 2 = 0) \\ \lceil x \rceil & else \end{cases}$$

convertToIntegerTowardZero:

$$roundTowardZero(x) = \begin{cases} \lfloor x \rfloor & \text{if } (x > 0) \\ \lceil x \rceil & else \end{cases}$$

convertToIntegerTowardPositive:

$$roundTowardPositive(x) = \lceil x \rceil$$

convertToIntegerTowardNegative:

$$roundTowardNegative(x) = \lfloor x \rfloor$$

convertToIntegerTiesToAway:

$$roundTiesToAway(x) = \begin{cases} \lfloor x \rfloor & \text{if } (x - \lfloor x \rfloor < 0.5) \\ \lceil x \rceil & \text{if } (x - \lfloor x \rfloor > 0.5) \\ \lceil x \rceil & \text{if } (x > 0) \\ \lfloor x \rfloor & else \end{cases}$$

convertToIntegerExactTiesToEven:

The rounding method $roundTiesToEven$ is used and if the result is unequal to x, the result is signaling.

convertToIntegerExactTowardZero:

The rounding method $roundTowardZero$ is used and if the result is unequal to x, the result is signaling.

convertToIntegerExactTowardPositive:

The rounding method $roundTowardPositive$ is used and if the result is unequal to x, the result is signaling.

convertToIntegerExactTowardNegative:

The rounding method $roundTowardNegative$ is used and if the result is unequal to x, the result is signaling.

convertToIntegerExactTiesToAway:

The rounding method $roundTiesToAway$ is used and if the result is unequal to x, the result is signaling.

4.2.4 Conversion operations

4.2.4.1 convertFromHexCharacter

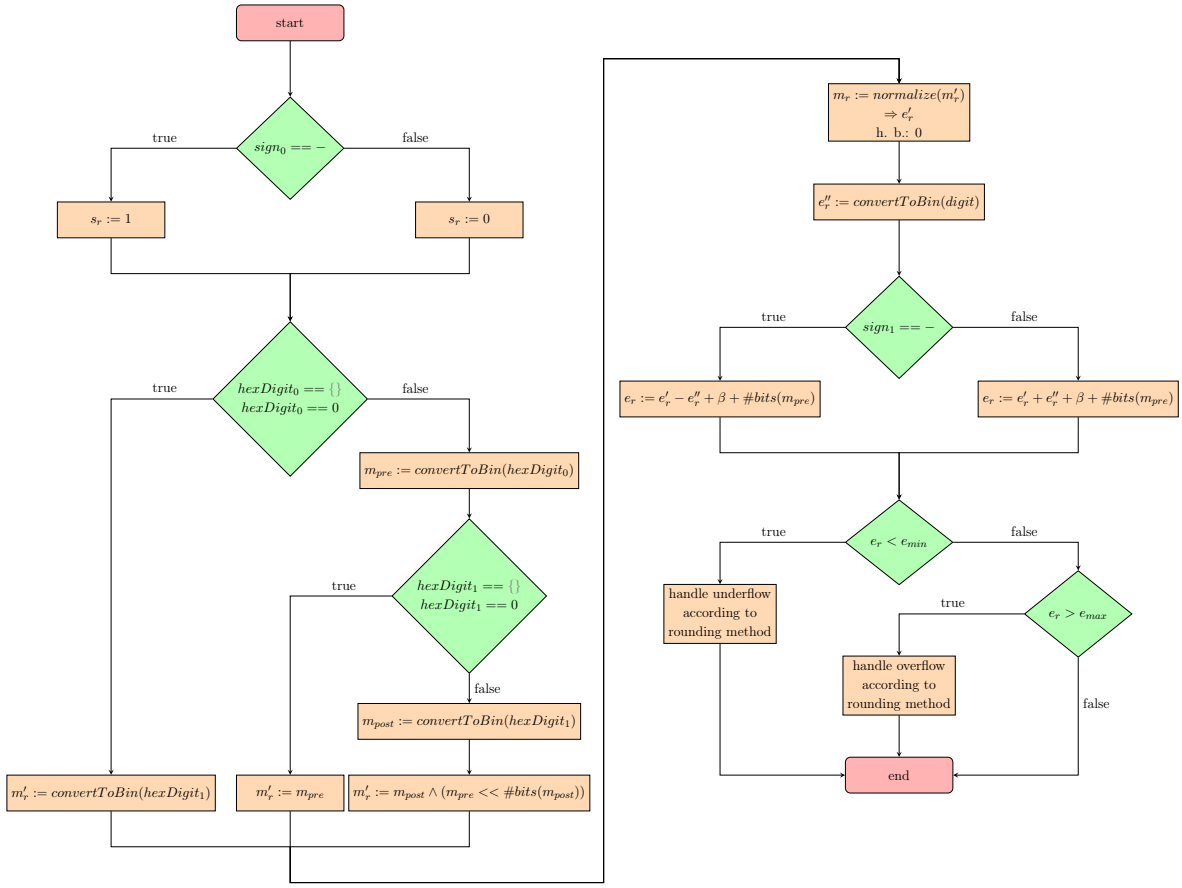

Figure 4.21: Definition of **convertFromHexCharacter**

The operation $convertFromHexCharacter$ defined in Figure 4.21 converts a given number in a hexadecimal representation into a floating point number. The hexadecimal representation is described by the grammar shown in Figure 4.22, taken from [11].

If the sign_0 is equal to $-$, the output has to be negative, so that s_r has to be one. Otherwise it has to be zero.

sign	[+−]
digit	[0123456789]
hexDigit	[0123456789abcdefABCDEF]
hexExpIndicator	[Pp]
hexIndicator	"0"[Xx]
hexSignificand	({hexDigit}* "." {hexDigit}+ | {hexDigit}+ "." | {hexDigit}+)
decExponent	{hexExpIndicator}{sign}?{digit}+
hexSequence	{$\underbrace{\text{sign}}_{sign_0}$}?{hexIndicator}{$\underbrace{\text{hexSignificand}}_{hexDigit_0, hexDigit_1}$}{$\underbrace{\text{decExponent}}_{sign_1, digit}$}

Figure 4.22: Definition of **hexSequence** [11]

According to the grammar shown in Figure 4.22 there are three different ways, the hexSignificand can look like: hexDigit_0 can be nonexistent when hexDigit_1 is existent, hexDigit_1 can be nonexistent when hexDigit_0 is existent or both can be existent. If hexDigit_0 is nonexistent, the not-normalized mantissa is equal to the binary representation of hexDigit_1. If hexDigit_0 is existent and hexDigit_1 is not, the not-normalized mantissa is equal to the binary representation of hexDigit_0. If both, hexDigit_0 and hexDigit_1, the not-normalized mantissa is equal to the binary representation of hexDigit_0 followed by the binary representation of hexDigit_1. The mantissa of the result is calculated by normalizing the not-normalized one.
The representation of decExponent is not biased-based as it is for floating point numbers, it makes use of a sign instead. If the sign_1 is equal to −, the binary representation of digit has to be subtracted, otherwise it has to be added during the calculation of the resulting exponent. The bias β, the normalization factor e_r' from the normalization of the mantissa and the number of bits required for the binary representation of hexDigit_0, what is zero if $\text{hexDigit}_0 == 0$ holds, have be added to calculate the resulting exponent.
Finally, if an underflow or overflow occurs, it has to be handled according to the selected rounding method.

4.2.4.2 convertToHexCharacter

The operation *convertToHexCharacter* defined in Figure 4.23 converts a given floating point number into a hexadecimal representation described by the following regular expression according to the grammar defined in [11] :

$$\underbrace{[+-]}_{sign_0}\overbrace{\text{"}0x\text{"}}^{hexIndicator}\underbrace{[01]}_{hexDigit_0}\text{"."}\overbrace{[0123456789abcdef]+}^{hexDigit_1}\underbrace{\text{"}p\text{"}}_{hexExpIndicator}\overbrace{[+-]}^{sign_1}\underbrace{[0123456789]+}_{digit}$$

A conversion to the hexadecimal representation is impossible, if the input is either NaN or $\pm\infty$, so that an exception has to be thrown if one of these cases occur.
If none of the special cases apply, *hexIndicator* and *hexExpIndicator* can be set because

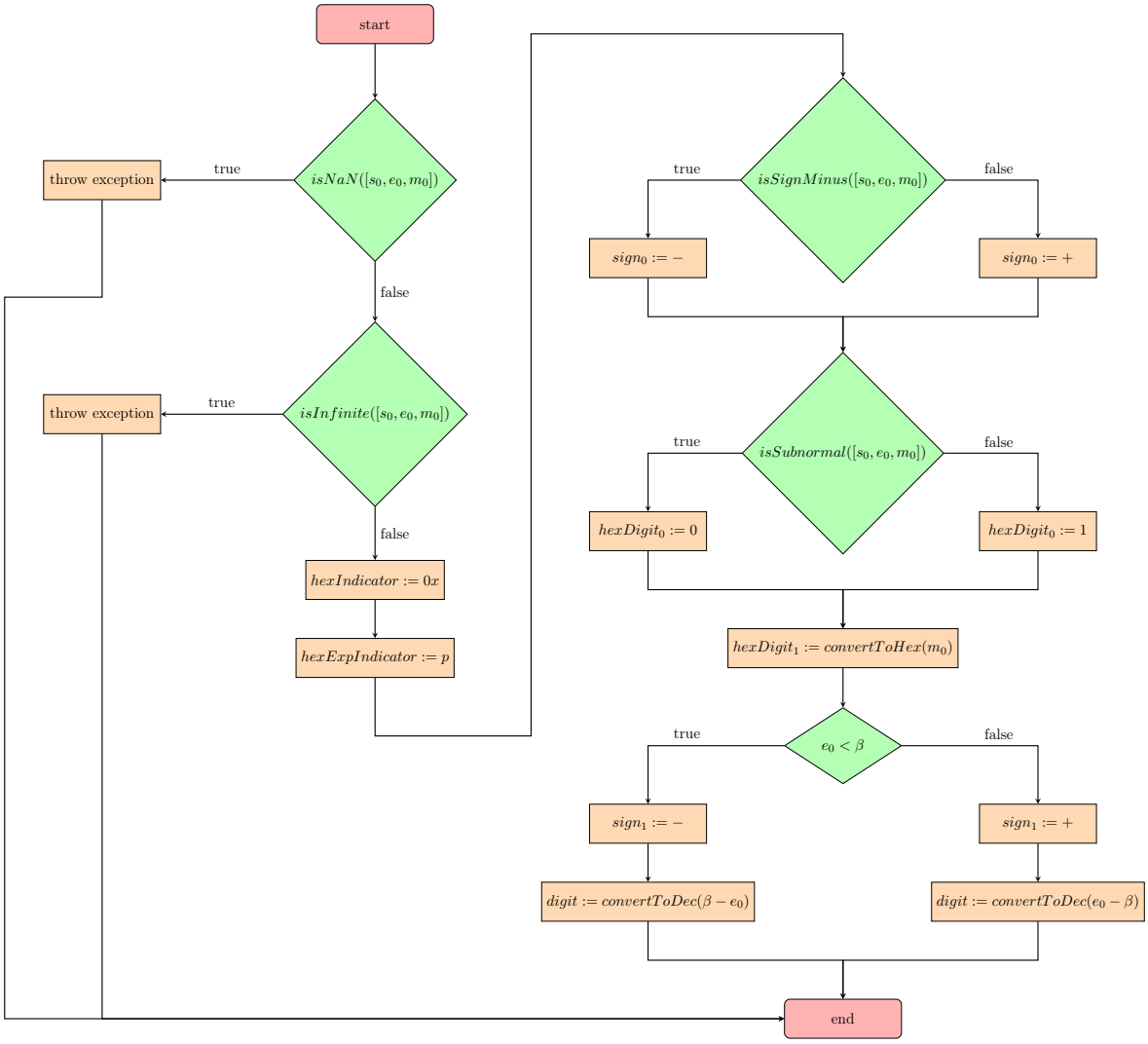

Figure 4.23: Definition of **convertToHexCharacter**

they are always the same, irrespective of the floating point input.
If the input is negative, the sign_0 of the output has to be set to "−" and if it is positive, it has to be set to "+" instead. In the next step, $hexDigit_0$, what is the pre-point digit, has to be determined. If the input is subnormal, it is zero, otherwise it is one. The post-point digits $hexDigit_1$, means the mantissa, can be simply calculated by converting the binary mantissa m_0 into a hexadecimal number. The hexadecimal representation of negative exponents is not bias-based, it is simply done with the usage of a sign. So if e_0 is smaller than the bias β, the exponent is negative and thus sign_1 is set to "-". Otherwise, sign_1 is set to "+". Finally, the decimal exponent is equal to the difference $\beta - e_0$ for negative or $e_0 - \beta$ for positive exponents.

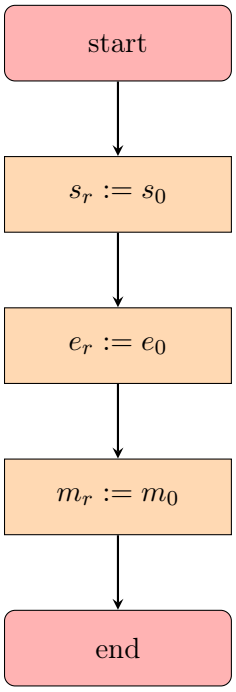

Figure 4.24: Definition of **copy**

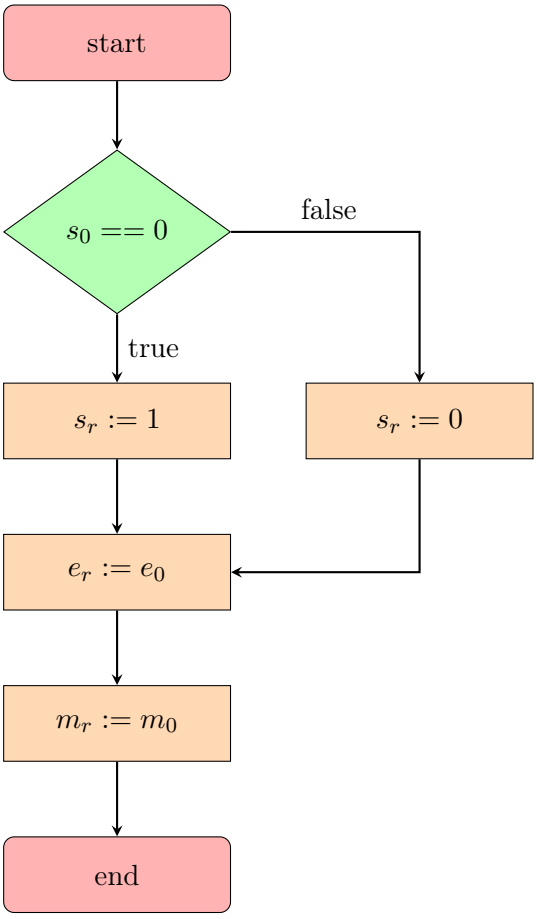

Figure 4.25: Definition of **negate**

4.2.5 Sign bit operations

4.2.5.1 copy

The operation *copy* forwards the input floating-point number to the output. So the definition depicted in Figure 4.24 is self-explanatory.

4.2.5.2 negate

To switch the sign of a given floating point number, the operation *negate* defined in Figure 4.25 is used. It changes the sign and forwards mantissa and exponent of the given input to the output.

4.2.5.3 abs

Similar to the operation *negate*, the operation *abs* shown in Figure 4.26 only changes the sign of a given input. But in that case, the sign is set to 0 independent from the input sign.

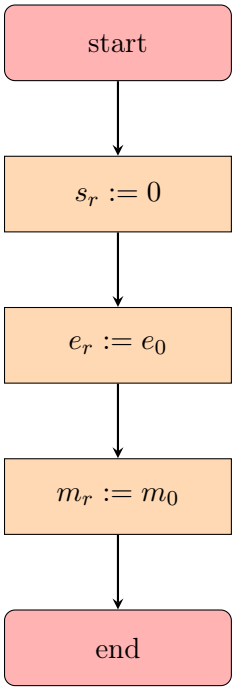

Figure 4.26: Definition of **abs**

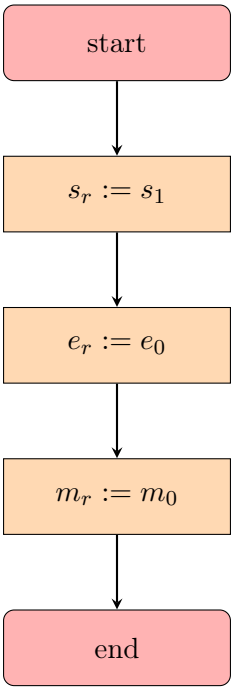

Figure 4.27: Definition of **copySign**

4.2.5.4 copySign

The function *copySign* defined in Figure 4.27 returns a floating point number consisting of the sign of the second input and mantissa and exponent of the first input.

4.2.6 Comparison operations

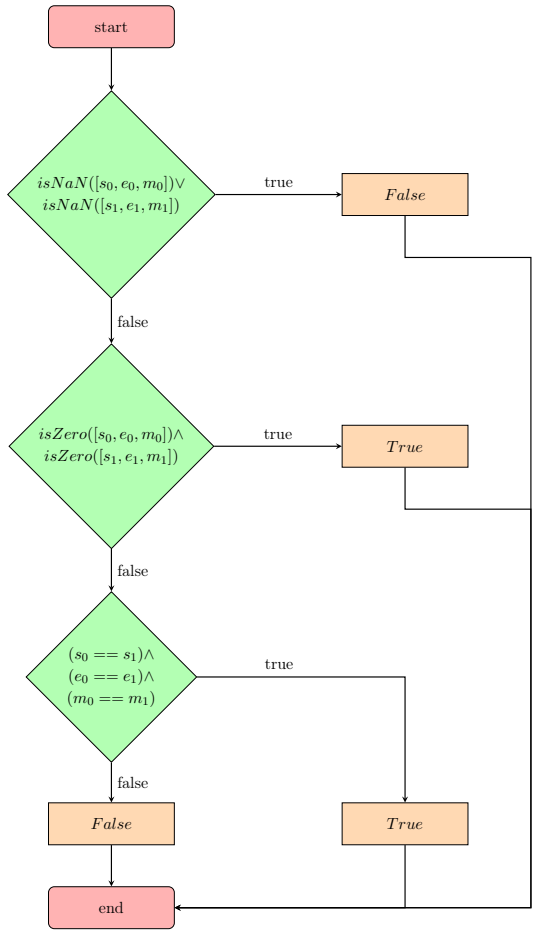

Figure 4.28: Definition of **compareQuietEqual**

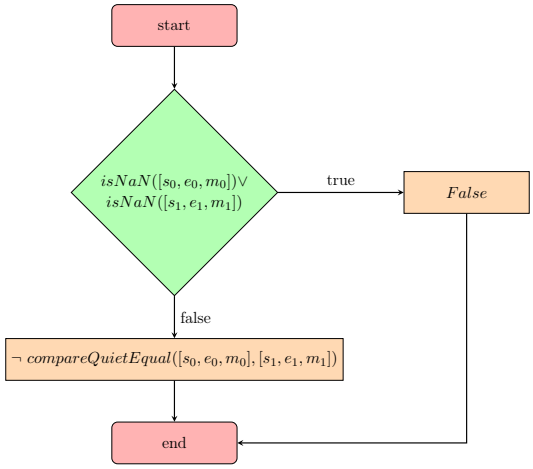

Figure 4.29: Definition of **compareQuietNotEqual**

4.2.6.1 compareQuietEqual

The operation *compareQuietEqual* defined in Figure 4.28 returns $True$ iff the binary representation of the two inputs $[s_0, e_0, m_0]$ and $[s_1, e_1, m_1]$ are identical or both inputs are zero. So the definition is self-explanatory: If at least one of the inputs is a NaN, $False$ is returned. Otherwise, the inputs are checked to be both equal to zero, the sign does not care in this case. If that comparison does not hold, the binary representations of the inputs are checked to be equal. If that holds, the result is $True$, otherwise it is $False$.

4.2.6.2 compareQuietNotEqual

To define the operation *compareQuietNotEqual* depicted in Figure 4.29, the result of the operation *compareQuietEqual* with the same inputs can be negated. But because both *compareQuietNotEqual* and *compareQuietEqual* return the result $False$ if at least one of the inputs is a NaN, that has to be checked before using *compareQuietEqual*.

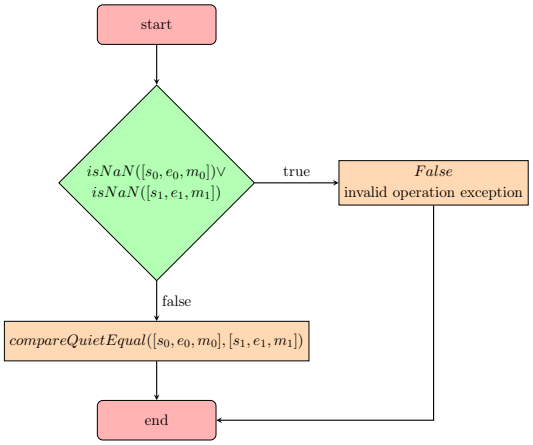

Figure 4.30: Definition of **compareSignalingEqual**

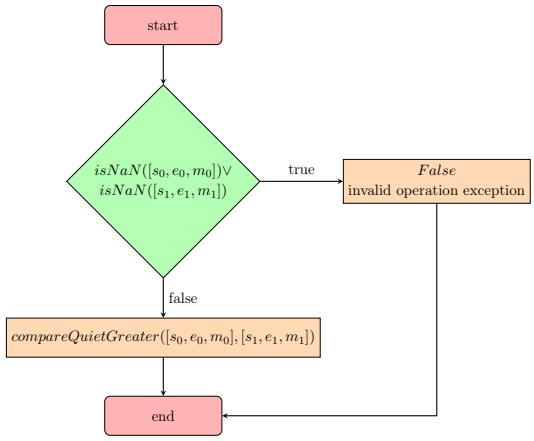

Figure 4.31: Definition of **compareSignalingGreater**

4.2.6.3 compareSignalingEqual

The operation *compareSignalingEqual* shown in Figure 4.30 is nearly identical with the operation *compareQuietEqual* defined above. The only difference it that an exception is raised if at least one of the inputs is a NaN. Because of this, the inputs are first checked to be NaN and, if that does not hold, the operation *compareQuietEqual* is used afterwards.

4.2.6.4 compareSignalingGreater

The operation *compareSignalingGreater* defined in Figure 4.31 behaves like the operation *compareQuietGreater* explained below except the fact, that at least one NaN as input leads to an additional exception. So the inputs are tested for being NaN and if that does not hold, the function *compareQuietGreater* is used.

4.2.6.5 compareSignalingGreaterEqual

The operation *compareSignalingGreaterEqual* depicted in Figure 4.32 returns the result $False$ and raises an exception if at least one of the inputs is a NaN. Otherwise, the output is equal to the output of the operation *compareQuietGreaterEqual* defined below. So this operation is then used to compute the result.

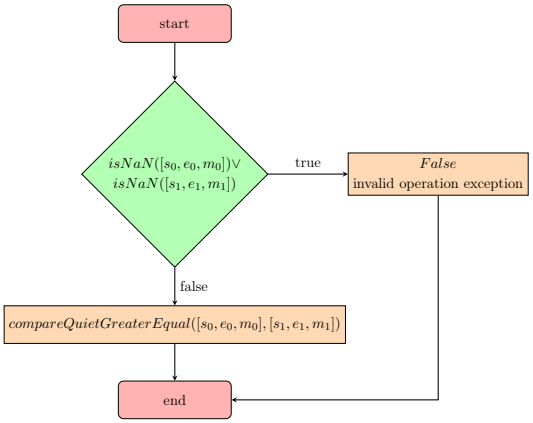

Figure 4.32: Definition of **compareSignaling-GreaterEqual**

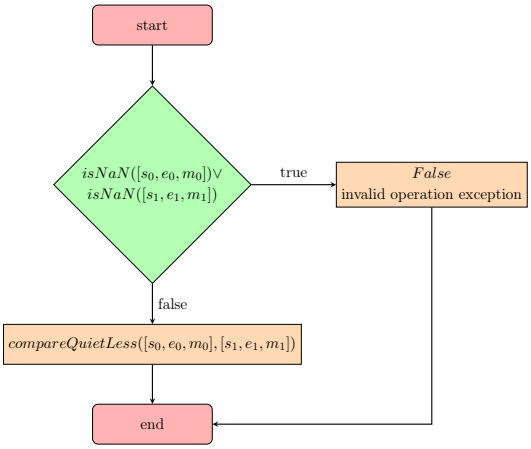

Figure 4.33: Definition of **compareSignalingLess**

4.2.6.6 compareSignalingLess

To define the operation *compareSignalingLess* shown in Figure 4.33, the operation *compareQuietLess* is used. The only difference is that if at least one of the inputs is a NaN, an exception is raised.

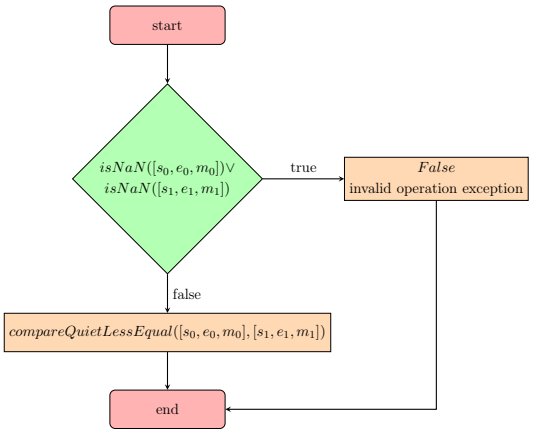

Figure 4.34: Definition of **compareSignalingLessEqual**

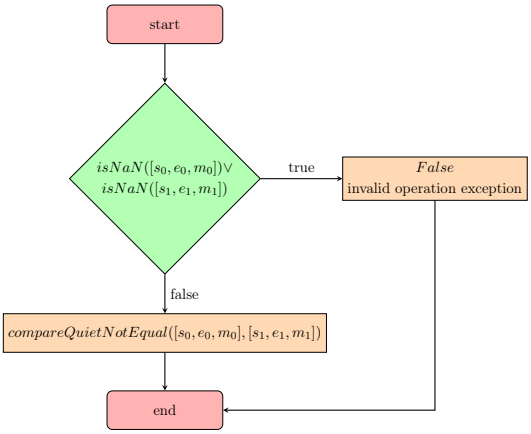

Figure 4.35: Definition of **compareSignalingNotEqual**

4.2.6.7 compareSignalingLessEqual

The operation *compareSignalingLessEqual* is defined in Figure 4.34. If at least one of the inputs is a NaN, the result $False$ is returned and an exception is raised. Otherwise, the

output is equal to the one of *compareQuietLess*, so this operation is used to calculate the result.

4.2.6.8 compareSignalingNotEqual

To define the operation *compareSignalingNotEqual* shown in Figure 4.35, the operation *compareQuietNotEqual* is used after neglecting the special case: If at least one of the inputs is a NaN, an exception is raised and the result of the comparison is set to $False$.

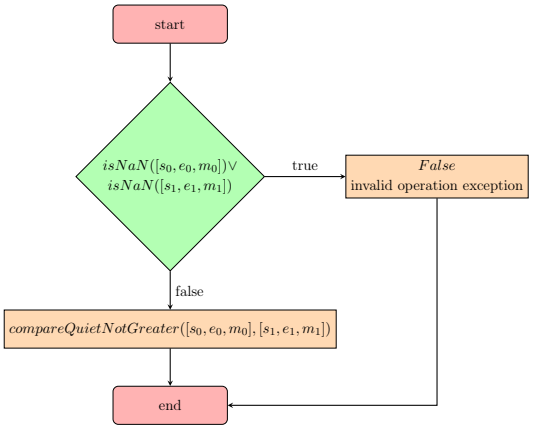

Figure 4.36: Definition of **compareSignaling-NotGreater**

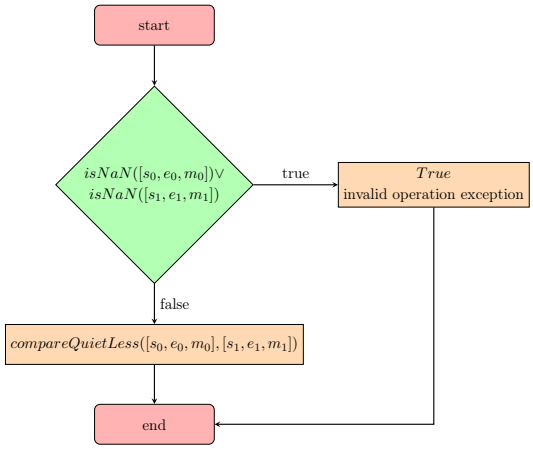

Figure 4.37: Definition of **compareSignaling-LessUnordered**

4.2.6.9 compareSignalingNotGreater

The operation *compareSignalingNotGreater* defined in Figure 4.36 raises an exception if at least one of the inputs is a NaN and sets the result to $False$. Otherwise, the result corresponds to the result of the operation *compareQuietNotGreater*, so it is used for calculating the output after checking for the occurrence of $NaNs$.

4.2.6.10 compareSignalingLessUnordered

The operation *compareSignalingLessUnordered* depicted in Figure 4.37 returns $True$ and raises an exception if at least one of the inputs is a NaN. Otherwise, the operation *compareQuietLess* is used to calculate the result.

4.2.6.11 compareSignalingNotLess

The operation *compareSignalingNotLess* defined in Figure 4.38 returns $False$ and raises an exception if at least one of the inputs is a NaN. If that is not the case, the operation *compareQuietNotLess* is used to compute the result of the comparison.

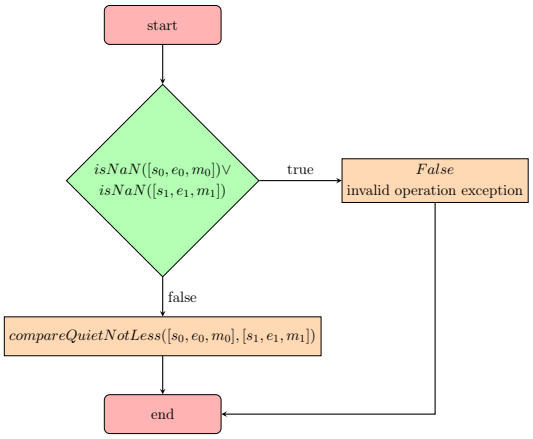

Figure 4.38: Definition of **compareSignalingNotLess**

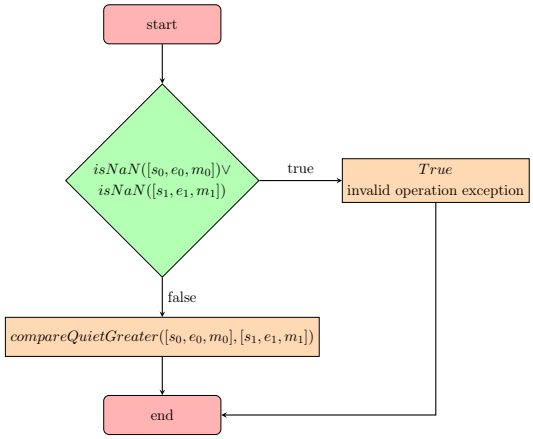

Figure 4.39: Definition of **compareSignaling-GreaterUnordered**

4.2.6.12 compareSignalingGreaterUnordered

The operation *compareSignalingGreaterUnordered* shown in Figure 4.39 returns $True$ and raises an exception if at least one of the inputs is a NaN. Apart from that, the operation *compareQuietGreater* provides the same results and so it is used for the further calculation.

4.2.6.13 compareQuietGreater

The operation *compareQuietGreater* defined in Figure 4.40 returns a NaN if at least one of the inputs is a NaN. Since $-0 == +0$ hold for comparisons in this arithmetic, $False$ is also returned if both inputs are zero, independent of the sign. In the next step, the sign is considered: If $s_0 == 1$ and $s_1 == 0$ holds, the first input is negative and the second one is positive, so the result of *compareQuietGreater* is $False$ in this case. If $s_0 == 0$ and $s_1 == 1$ holds in other ways, the result is $True$, no matter how the exponents and mantissas look like because $\forall x \in \Re, x \neq 0 : +|x| > -|x|$ holds. If both signs are equal, it has to be distinguished between positive and negative inputs.
If the inputs are both positive, the result is $True$ if the exponent of the first input is greater than the one of the second input ($e_0 > e_1$). If $e_0 < e_1$ holds, the result is $False$. The mantissas are only considered, if the signs as well as the exponents of both inputs are equal to each other. In that case, the result corresponds to the comparison $m_0 > m_1$.
If the inputs are both negative, the result behaves to the contrary of the result with two positive inputs. If the first exponent is greater than the second one ($e_0 > e_1$), then the output is $False$. Otherwise, if $e_0 < e_1$ holds, the result is $True$. If both, the signs and the exponents are equal, the mantissas are used to determine the result.

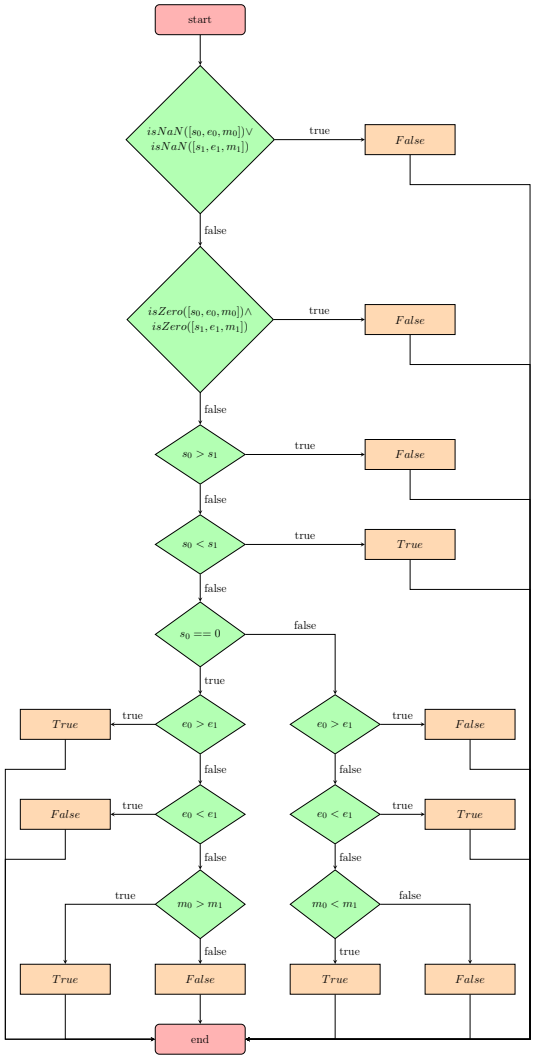

Figure 4.40: Definition of **compareQuietGreater**

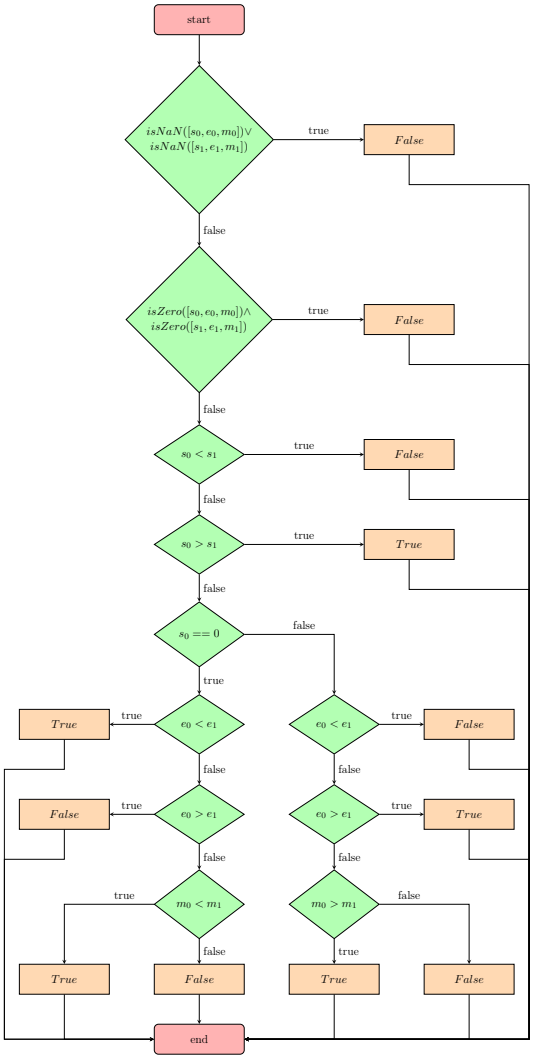

Figure 4.41: Definition of **compareQuietLess**

4.2.6.14 compareQuietGreaterEqual

The operation *compareQuietGreaterEqual* defined in Figure 4.42 returns $False$ if at least one of the inputs is a NaN. If that special case does not occur, the disjunction of the operations *compareQuietEqual* and *compareQuietGreater* is used to determine the result.

4.2.6.15 compareQuietLess

The operation *compareQuietLess* depicted in Figure 4.41 returns $False$ if at least one of the inputs is a NaN. Because $+0 == -0$ holds in this arithmetic, the result is $False$ too, if both inputs are zero independent of the sign. The next step is to consider the signs. If $s_0 < s_1$, which means the first input is positive and the second one negative, holds, the result is $False$.

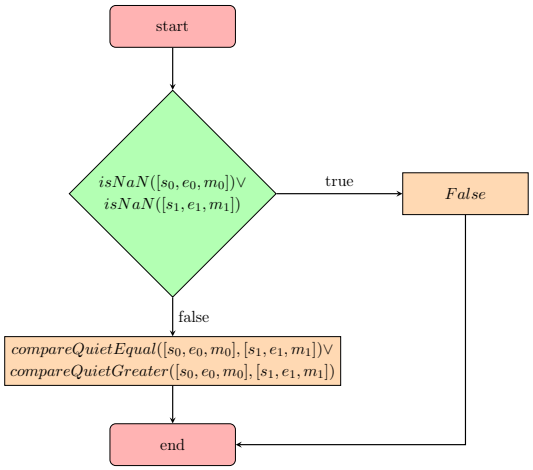

Figure 4.42: Definition of **compareQuietGreaterEqual**

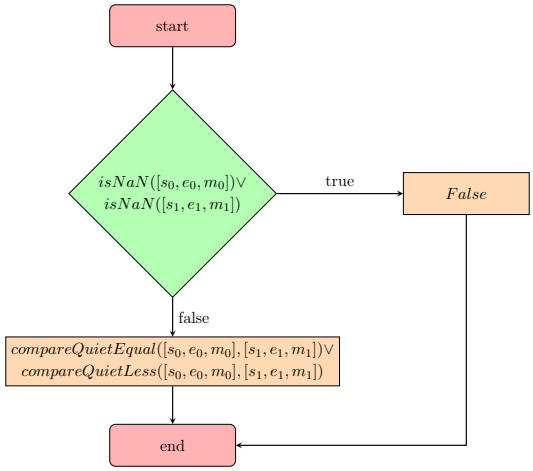

Figure 4.43: Definition of **compareQuietLessEqual**

If the opposite holds, the first input is negative and the second one is positive ($s_0 > s_1$), the result is $True$. The exponents are required to distinguish the result iff both signs are equal. If both inputs are positive, $e_0 < e_1$ leads to $True$ as output, while $e_0 > e_1$ leads to a $False$. If the exponents are also equal to each other, the mantissas become necessary for the comparison. If $m_0 < m_1$ holds, the result is $True$ and apart from that it is $False$.
If both inputs are negative, $e_0 < e_1$ leads to $False$ as ouput and the operation returns $True$ if $e_0 > e_1$ holds. Again, the mantissas are only considered, if the signs as well as the exponents are equal. $m_0 < m_1$ results in a $False$ as output while $m_0 \geq m_1$ results in a $True$.

4.2.6.16 compareQuietLessEqual

The operation *compareQuietLessEqual* defined in Figure 4.43 returns $False$ if at least one of the inputs is a NaN. Otherwise, the disjunction of the operations *compareQuietEqual* and *compareQuietLess* is utilized to ascertain the result of the comparison.

4.2.6.17 compareQuietUnordered

The operation *compareQuietUnordered* defined in Figure 4.44 returns $True$ if the result of a comparison is undefined, which means that at least one of the inputs has to be NaN. Otherwise, it returns $False$.

4.2.6.18 compareQuietNotGreater

Figure 4.45 shows the definition of the operation *compareQuietNotGreater*. It returns $False$ if at least one of the inputs is a NaN. Otherwise the negation of the operation *compareQuietGreater* is used to compute the result.

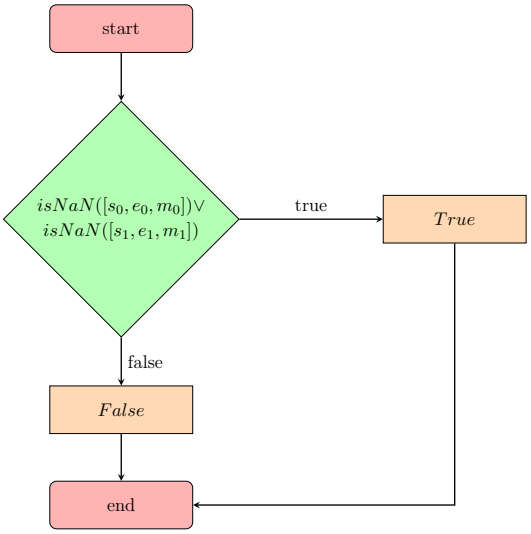

Figure 4.44: Definition of **compareQuietUnordered**

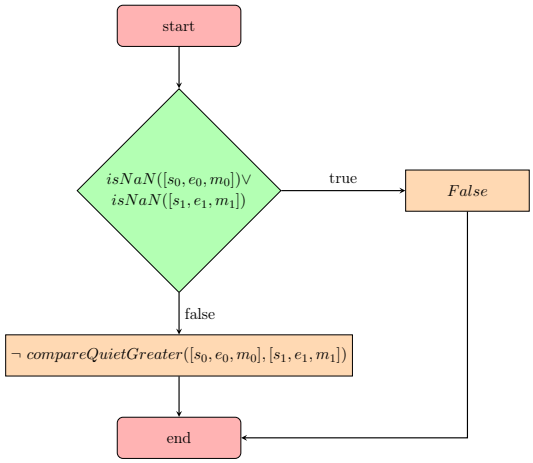

Figure 4.45: Definition of **compareQuietNotGreater**

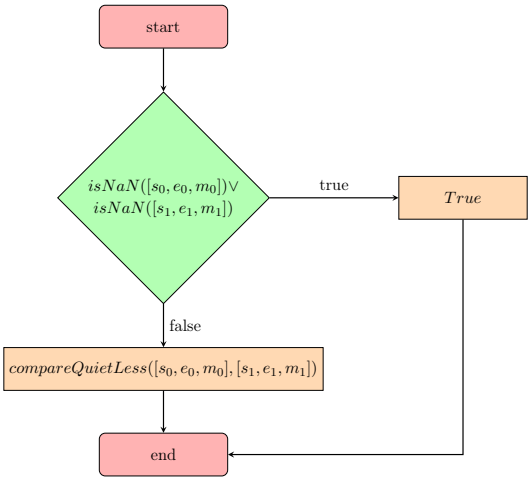

Figure 4.46: Definition of **compareQuietLessUnordered**

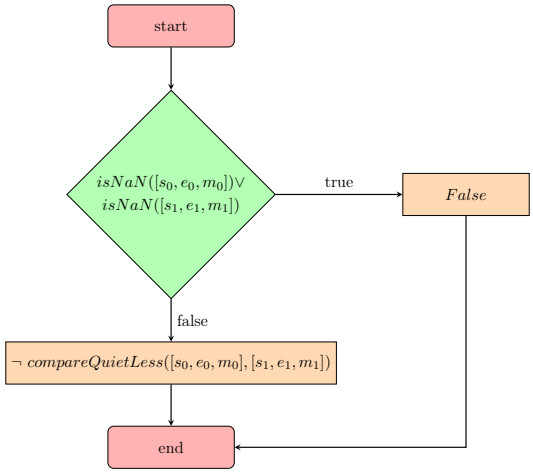

Figure 4.47: Definition of **compareQuietNotLess**

4.2.6.19 compareQuietLessUnordered

The operation *compareQuietLessUnordered* defined in Figure 4.46 sets $True$ as output if at least one of the inputs is a NaN. Apart from that, the operation *compareQuietLess* provides the result of the comparison.

4.2.6.20 compareQuietNotLess

The operation *compareQuietNotLess* shown in Figure 4.47 returns $False$ if at least one of the inputs is a NaN. Otherwise, the negation of the operation *compareQuietLess* supplies the desired result.

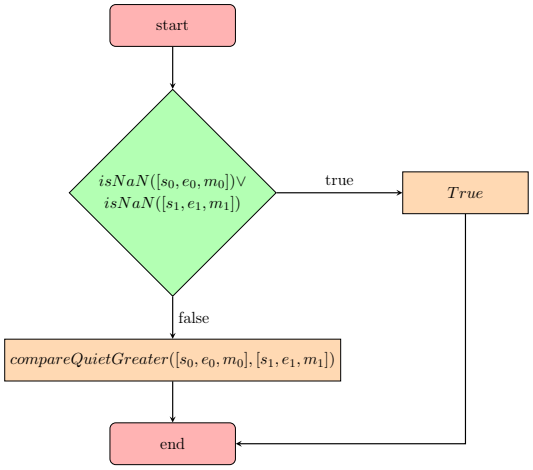

Figure 4.48: Definition of **compareQuietGreaterUnordered**

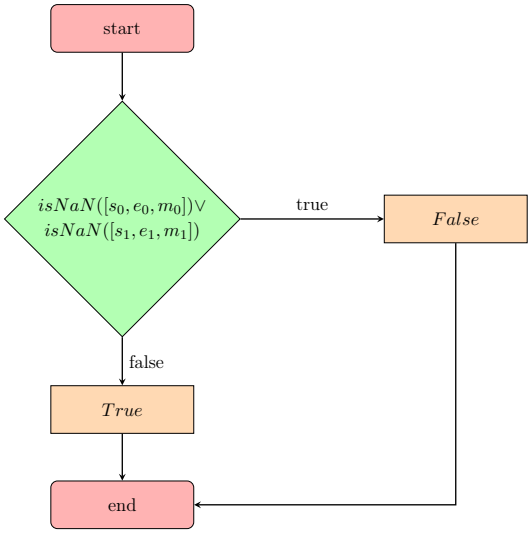

Figure 4.49: Definition of **compareQuietOrdered**

4.2.6.21 compareQuietGreaterUnordered

Figure 4.48 shows the definition of the operation *compareQuietGreaterUnordered*. It returns $True$ if at least one of the inputs is a NaN. Otherwise, the output corresponds to the output of the operation *compareQuietGreater*.

4.2.6.22 compareQuietOrdered

The operation *compareQuietOrdered* depicted in Figure 4.49 returns $False$ if at least one of the inputs is a NaN and the result of a comparison would be undefined. In all other cases, it returns $True$.

4.2.7 Conformance predicates

4.2.7.1 is754version1985

The arithmetic ought to be conform to the outdated IEEE Standard 754-1985 [10], but to be on the safe side, the function $is754version1985$ returns $false$.

4.2.7.2 is754version2008

This thesis defines an IEEE conform floating point arithmetic. The currently most actual one is the IEEE Standard 754-2008 [11], so the definitions are based on the specifications formulated in that document.
For this reason, the function $is754version2008$ has to return $true$ obviously.

4.2.8 General non-computational operations

4.2.8.1 class

The function *class* defined in Figure 4.50 returns the category, the input belongs to. There are ten different categories: $signalingNaN$, $quietNaN$, $negativeInfinity$, $negativeNormal$, $negativeSubnormal$, $negativeZero$, $positiveZero$, $positiveSubnormal$, $positiveNormal$ and $positiveInfinity$. If none of these categories fit to the input, the output is $undefined$ but that case cannot occur if the implementation is correct.

4.2.8.2 isSignMinus

The function $isSignMinus$ defined in Figure 4.51 returns $True$ iff the inputs sign bit is equal to 1.

4.2.8.3 isNormal

The function $isNormal$ depicted in Figure 4.52 returns $True$ iff the input is normal, which means the value exponent has to be $1 \leq e \leq 2^W - 2$.

4.2.8.4 isFinite

The function $isFinite$ defined in Figure 4.53 returns $True$ iff the input is a floating point number that is not NaN or ∞ neither. So the result is $True$ iff the input is zero, a subnormal or a normal. So the disjunction of the functions $isZero$, $isSubnormal$ and $isNormal$ are used to determine the result.

4.2.8.5 isZero

The function $isZero$ shown in Figure 4.54 returns $True$ iff the input is equal to zero, independent of the sign. According to the definition an input is zero iff the exponent as well as the mantissa are equal to zero.

4.2.8.6 isSubnormal

The function $isSubnormal$ defined in Figure 4.55 returns $True$ iff the input is a subnormal. That is the case iff the exponent is equal to zero while the mantissa is unequal to zero.

4.2.8.7 isInfinite

The function $isInfinite$ defined in Figure 4.56 returns $True$ iff the input is a floating point number equal to $\pm\infty$. According to the definition that is the case iff the exponent is equal to $2^W - 1$ and the mantissa is equal to zero.

4.2.8.8 isNaN

The function $isNaN$ shown in Figure 4.57 returns $True$ iff the input is NaN, independent of being signaling ($sNaN$) or quiet ($qNaN$).

4.2.8.9 isSignaling

The function $isSignaling$ depicted in Figure 4.58 returns $True$ iff the input is $sNaN$.

4.2.8.10 isCanonical

The function $isCanonical$ returns $True$ iff the representation of the input is canonical. Theoretically, $NaNs$ are not canonical because the mantissa only has to be non-zero, but in this definition the mantissa is always equal to 1 for $NaNs$ and so the function $isCanonical$ always returns $True$.

4.2.8.11 radix

The function $radix$ tells the radix of the format, which is 2 here.

4.2.8.12 totalOrder

The function *totalOrder* shown in Figure 4.59 is defined according to the definition of the IEEE Standard 754-2008 [11]. If the first input is smaller than the second input, the result is $True$. The function *compareQuietLess* is used to determine if that is the case. If the first input is larger than the second one, which is ascertained by using *compareQuietGreater*, the result is $False$.
If the first input is equal to the second one, which means that *compareQuietEqual* results in $True$, four different cases have to be distinguished: If both inputs are equal to zero, the first one with a negative sign and the second one with a positive sign, the result is $True$. If both inputs are equal to zero and the sign of the first input is positive while the sign of the second one is negative, the result of *totalOrder* is $False$. If the inputs are both negative and represent the same floating-point value, the result is $True$ iff the exponent of the first input is greater or equal to the exponent of the second input ($e_0 \geq e_1$). If both inputs represent the same floating-point value and do not fit to the previously mentioned combinations, *totalOrder* returns $True$ iff the exponent of the first input is less or equal to the exponent of the second input ($e_0 \leq e_1$).
If at least one of the inputs is a NaN, three different cases are differentiated: If the first input is a NaN with a negative sign while the second input is a floating point number, the result

of *totalOrder* is $True$. If the first input is a floating-point number and the second input is a NaN with a positive sign, the result is also $True$. If both inputs are $NaNs$, the IEEE Standard 754-2008 [11] specifies three rules: "negative sign orders below positive sign" [11], "signaling orders below quiet for $+NaN$, reverse for $-NaN$" [11] and "lesser payload, when regarded as an integer, orders below greater payload for $+NaN$, reverse for $-NaN$" [11]. So does the definition in this thesis.

4.2.8.13 totalOrderMag

The function $totalOrderMag$ shown in Figure 4.60 returns $totalOrder(abs(x), abs(y))$ as defined in the IEEE Standard 754-2008 [11].

4.2.9 Flag operations

4.2.9.1 lowerFlags

The operation $lowerFlags$ lowers the flags of the exceptions specified in the input *exceptionGroup* by setting the corresponding bits to zero.

4.2.9.2 raiseFlags

The operation $raiseFlags$ raises the flags of the exceptions specified in the input *exceptionGroup* by setting the corresponding bits to one.

4.2.9.3 testFlags

The operation $testFlags$ tests the flags of the exceptions specified in the input *exceptionGroup* by checking if the corresponding bits are set to one. If at least one of the bits is set, $true$ is returned, otherwise $false$ is returned.

4.2.9.4 testSavedFlags

The operation $testSavedFlags$ tests the flags in the input $flags$ of the exceptions specified in the input *exceptionGroup* by checking of the corresponding bits are set to one. If at least one of the bits is set, $true$ is returned, otherwise $false$ is returned.

4.2.9.5 restoreFlags

The operation $restoreFlags$ restores the status of the flags specified in the input *exceptionGroup* and stored in $flags$.

4.2.9.6 saveAllFlags

The operation $saveAllFlags$ "returns a representation of the state of all status flags" [11].

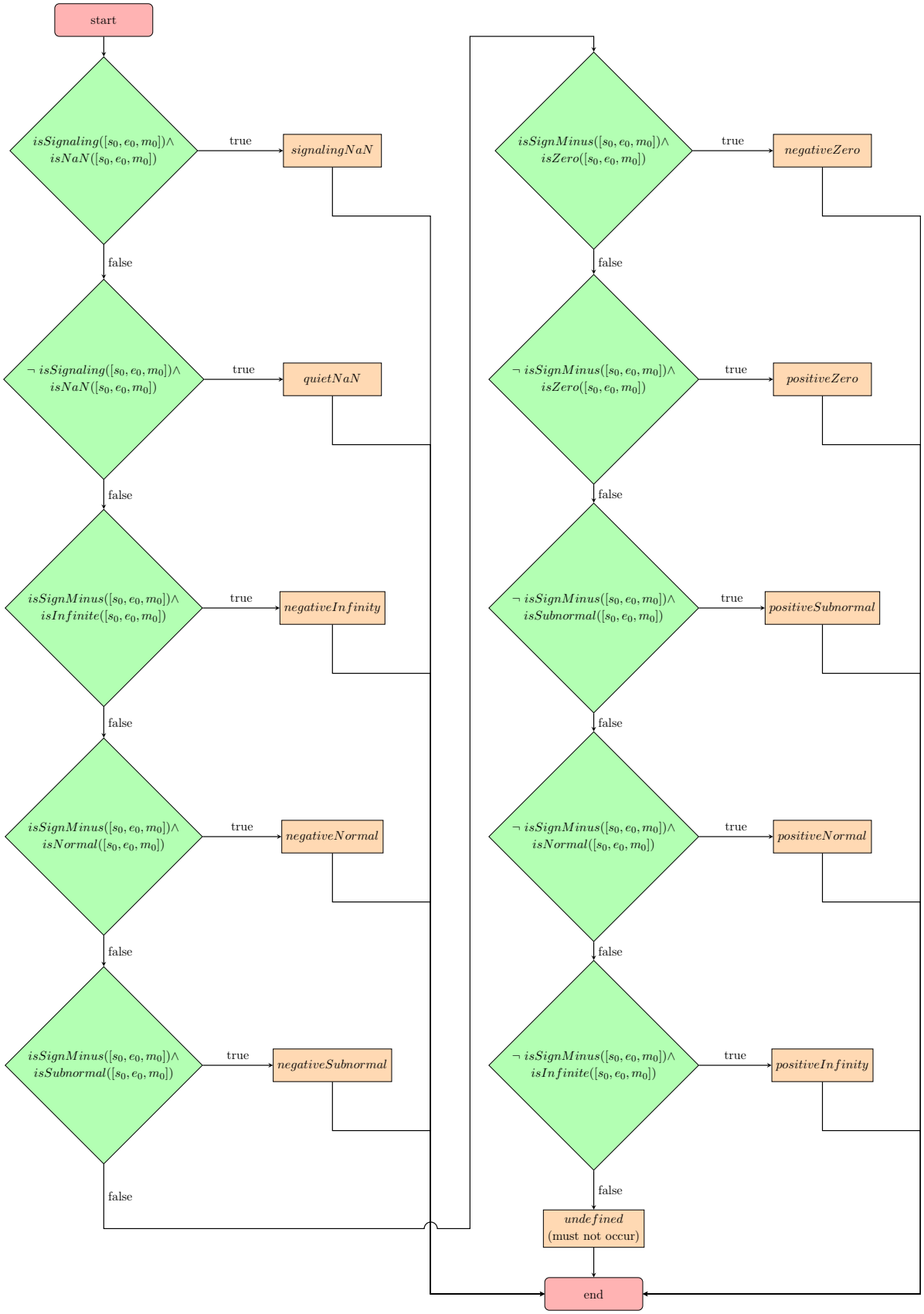

Figure 4.50: Definition of **class**

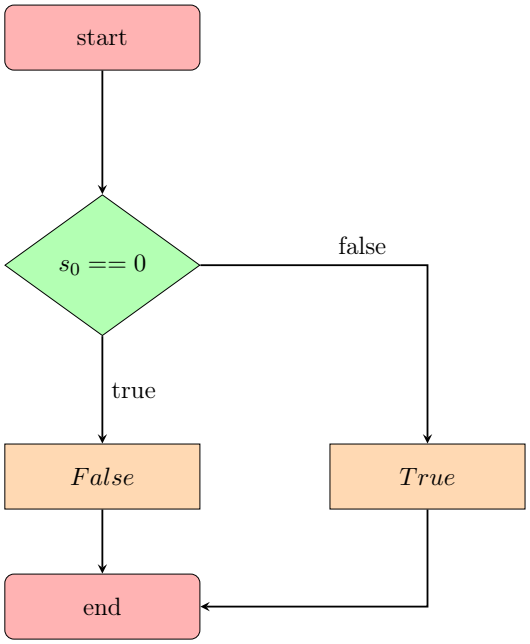

Figure 4.51: Definition of **isSignMinus**

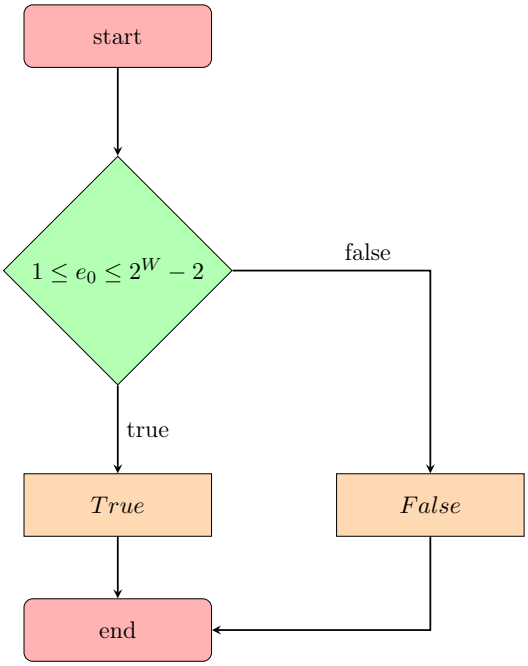

Figure 4.52: Definition of **isNormal**

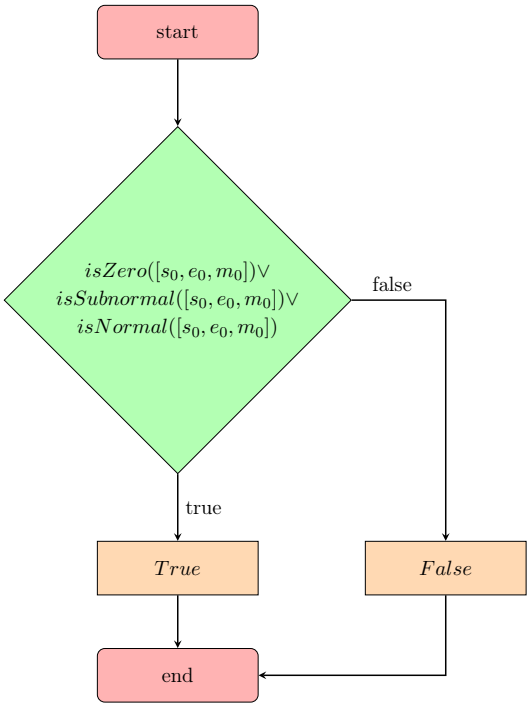

Figure 4.53: Definition of **isFinite**

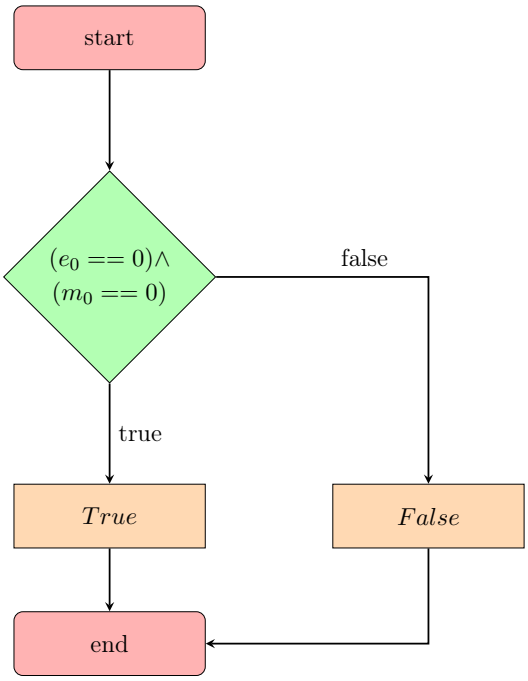

Figure 4.54: Definition of **isZero**

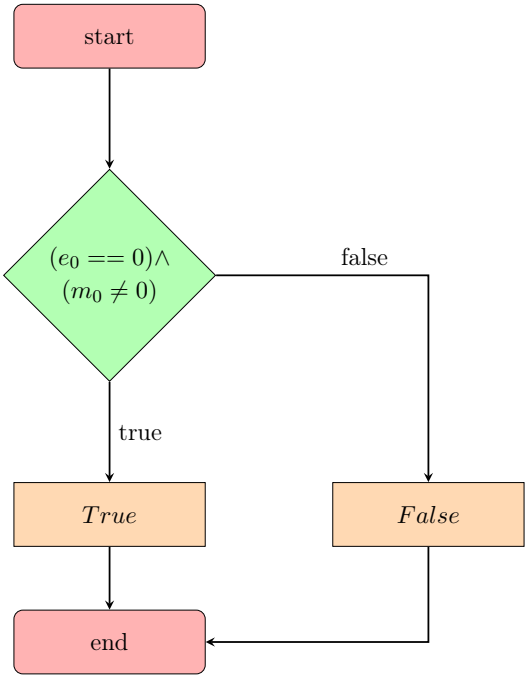

Figure 4.55: Definition of **isSubnormal**

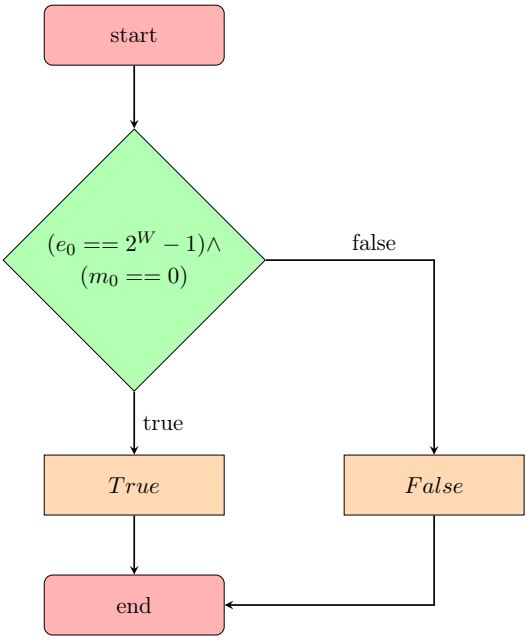

Figure 4.56: Definition of **isInfinite**

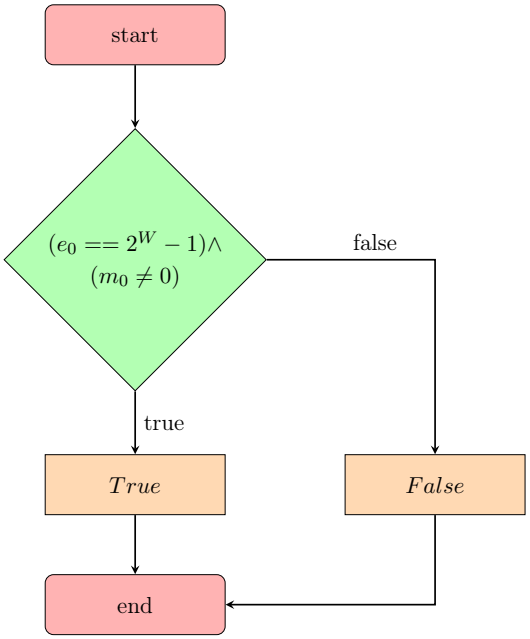

Figure 4.57: Definition of **isNaN**

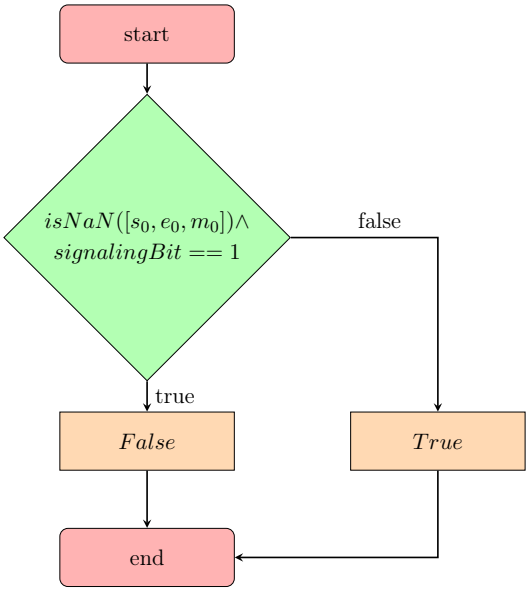

Figure 4.58: Definition of **isSignaling**

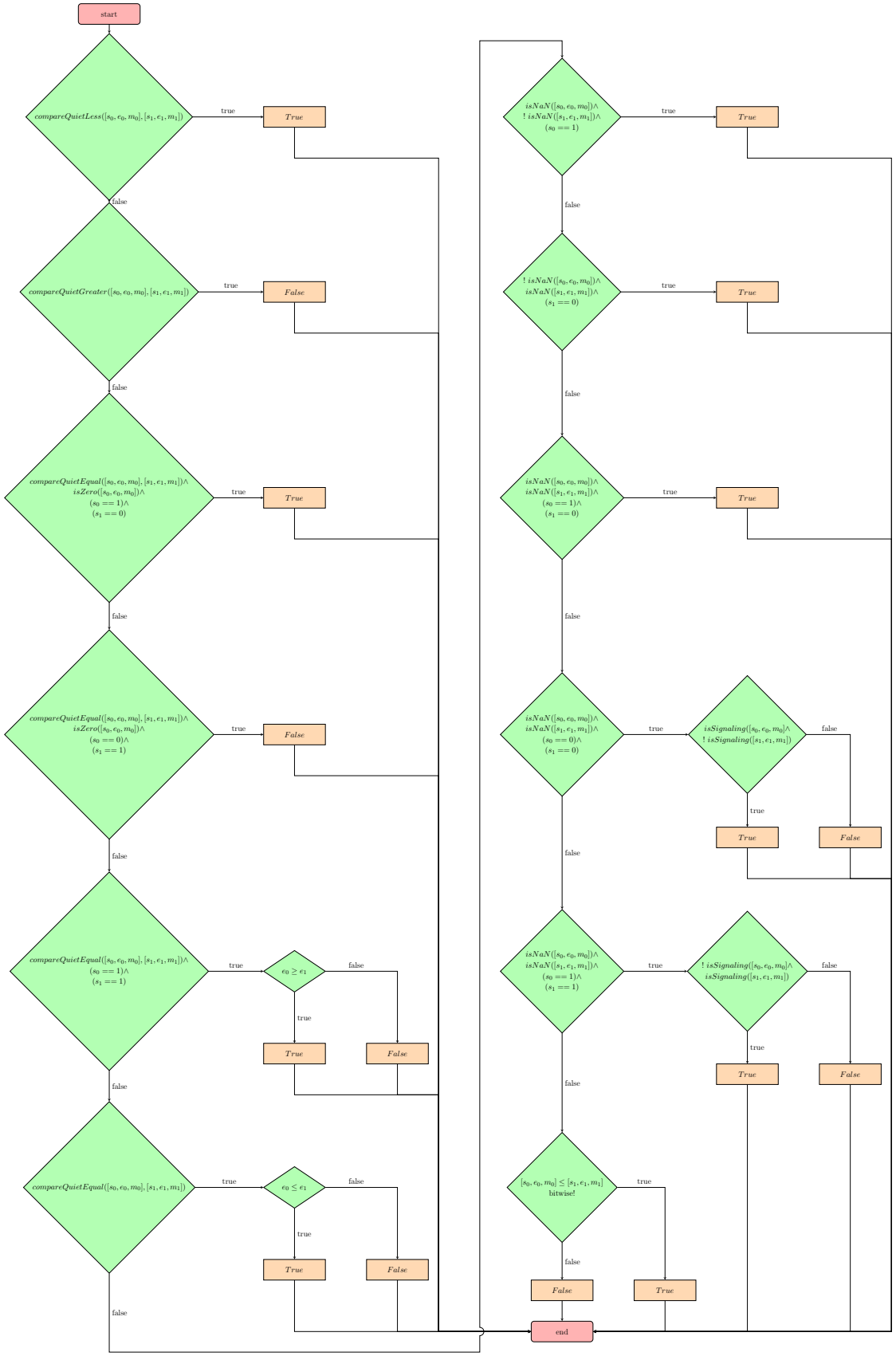

Figure 4.59: Definition of **totalOrder**

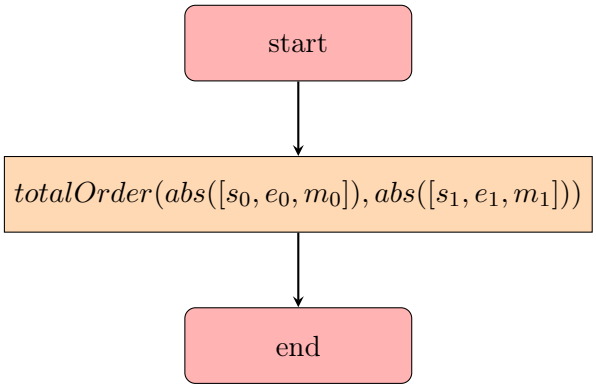

Figure 4.60: Definition of **totalOrderMag**

5 Verification

The definitions presented in Section 4.2 are verified in this chapter.
The groups conversion operations from Subsection 4.2.4, conformance predicates from Subsection 4.2.7, general non-computational operations from Subsection 4.2.8 and flag operations from Subsection 4.2.9 are not verified because they are completely defined in the IEEE Standard 754-2008 and therefore a verification is not required.
Special cases are explicitly omited during the verification because they are handled as it is standardized in the IEEE Standard 754-2008 and therefore they are irrelevant during the verification.

5.1 General computational operations

5.1.1 roundToIntegralTies*

The operations $roundToIntegralTiesToEven$ and $roundToIntegralTiesToAway$ always round the input to the nearest integral floating point number. Halfway cases are rounded to the next even integral floating point number if $roundToIntegralTiesToEven$ is selected or to the nearest integral floating point number away from zero if $roundToIntegralTiesToAway$ is selected. That are exactly the requirements defined in the IEEE Standard, therefore the definition is conform to the IEEE Standard 754-2008.

5.1.2 roundToIntegralToward*

The operation $roundToIntegralTowardZero$ always rounds the input to the nearest integral floating point number towards zero, $roundToIntegralTowardPositive$ always rounds towards $+\infty$ and $roundToIntegralTowardNegative$ always rounds towards $-\infty$. That are exactly the requirements defined in the IEEE Standard, therefore the definition is conform to the IEEE Standard 754-2008.

5.1.3 roundToIntegralExact

The operations $roundToIntegralExact$ raises an inexact exception if the input is not equal to the rounded result according to the IEEE Standard 754-2008. The rounding operations it makes use of are proven to be correct in Section 5.1.1 and Section 5.1.2 and the operations used for the comparison are verified in Section 5.5.1 and Section 5.5.3. Therefore $roundToIntegralExact$ is verified implicitly.

5.1.4 nextUp

Theorem 1. *Let x be a floating point number and nextUp(x) the output of the operation defined in Figure 4.4 with x as input. If no special cases occur the following holds*

$$\forall x \in FP \nexists y \in FP : (x < y) \wedge (y < nextUp(x)).$$

Proof. To prove Theorem 1, reductio ad absurdum is used. The claim

$$\exists x \in FP \exists y \in FP : (x < y) \wedge (y < nextUp(x))$$

is proven to be wrong.

Case 1: $(x == 0)$

$$\begin{aligned} & (x < y) \wedge (y < nextUp(x)) \\ \Leftrightarrow\quad & (0 < y) \wedge (y < +2^{e_{min}} \cdot 0.\underbrace{0...0}_{m-1}1 \quad | \cdot \tfrac{1}{2^{e_{min}}};\ << m \\ \Leftrightarrow\quad & 0 < m_y < 1,\ m_y \in \mathbb{N}_{binary} \qquad \lightning \textit{Case 1} \end{aligned}$$

Case 2: $(x > 0) \wedge (x$ subnormal$)$

$$\begin{aligned} & (x < y) \wedge (y < nextUp(x)) \qquad | x = +2^{e_{min}} \cdot 0.m_x \\ \Leftrightarrow\quad & (x < y) \wedge (y < nextUp(+2^{e_{min}} \cdot 0.m_x)) \quad | \cdot \tfrac{1}{2^{e_{min}}} \end{aligned}$$

Case 2.1: $(0.m_x < 0.m_y) \wedge (0.m_y < 0.m_{nextUp(x)}),\ m_{nextUp(x)} = m_x + 1$

$$\begin{aligned} & 0.m_x < 0.m_y < 0.m_{nextUp(x)} \qquad | << m \\ \Leftrightarrow\quad & m_x < m_y < m_x + 1,\ m_y \in \mathbb{N}_{binary} \qquad \lightning \textit{Case 2.1} \end{aligned}$$

Case 2.2: $(0.m_x < 0.m_y) \wedge (0.m_y < 1.0)$

$$\begin{aligned} & 0.m_x < 0.m_y < 1.0 \qquad | << m \\ \Leftrightarrow\quad & \underbrace{1...1}_{m} < m_y < 1\underbrace{0...0}_{m},\ m_y \in \mathbb{N}_{binary} \qquad \lightning \textit{Case 2.2} \end{aligned}$$

Case 3: $(x > 0) \wedge (x$ normal$)$

Case 3.1: $(x < y) \wedge (y < 1.m_{nextUp(x)}),\ m_{nextUp(x)} = m_x + 1$

$$\begin{aligned} & 2^{e_x-\beta} \cdot 1.m_x < 2^{e_y-\beta} \cdot 1.m_y < 2^{e_x-\beta} \cdot 1.m_{nextUp(x)} \quad | normalized \Rightarrow e_y = e_x \\ \Rightarrow\quad & 2^{e_x-\beta} \cdot 1.m_x < 2^{e_x-\beta} \cdot 1.m_y < 2^{e_x-\beta} \cdot 1.m_{nextUp(x)} \quad | \cdot \tfrac{1}{2^{e_x-\beta}} \\ \Leftrightarrow\quad & 1.m_x < 1.m_y < 1.m_{nextUp(x)} \qquad | -1;\ << m \\ \Leftrightarrow\quad & m_x < m_y < m_x + 1,\ m_y \in \mathbb{N}_{binary} \qquad \lightning \textit{Case 3.1} \end{aligned}$$

Case 3.2: $(x < y) \wedge (y < 10.0)$

$$\begin{aligned} & 2^{e_x-\beta} \cdot 1.m_x < 2^{e_y-\beta} \cdot 1.m_y < 2^{e_x-\beta} \cdot 10.0 \quad | normalized \Rightarrow e_y = e_x \\ \Rightarrow\quad & 2^{e_x-\beta} \cdot 1.m_x < 2^{e_x-\beta} \cdot 1.m_y < 2^{e_x-\beta} \cdot 10.0 \quad | \cdot \tfrac{1}{2^{e_x-\beta}} \\ \Leftrightarrow\quad & 1.m_x < 1.m_y < 10.0 \qquad | m_x = 10.0 - 0.\underbrace{0...0}_{m-1}1 = 1.\underbrace{1...1}_{m} \\ \Rightarrow\quad & 1.\underbrace{1...1}_{m} < 1.m_y < 10.0 \qquad | -1;\ << m \\ \Leftrightarrow\quad & \underbrace{1...1}_{m} < m_y < 1\underbrace{0...0}_{m},\ m_y \in \mathbb{N}_{binary} \qquad \lightning \textit{Case 3.2} \end{aligned}$$

Case 4: $(x < 0) \wedge (x \text{ subnormal})$

Case 4.1: $nextUp(x)$ subnormal

$$
\begin{aligned}
& e_x = e_{nextUp(x)} \Rightarrow e_y = e_x \\
& -2^{e_{min}} \cdot 0.m_x < -2^{e_{min}} \cdot 0.m_y < -2^{e_{min}} \cdot 0.m_{nextUp(x)},\ m_{nextUp(x)} = m_x - 1 \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \cdot \tfrac{1}{2^{e_{min}}} \\
\Leftrightarrow\ & -m_x < -m_y < -(m_x - 1) \qquad\qquad\qquad\qquad \mid \cdot (-1) \\
\Leftrightarrow\ & m_x > m_y > m_x - 1,\ m_y \in \mathbb{N}_{binary} \qquad\qquad\qquad\qquad \lightning \textit{Case } 4.1
\end{aligned}
$$

Case 4.2: $nextUp(x) = 0$

$$
\begin{aligned}
& x = -2^{e_{min}} \cdot 0.\underbrace{0...0}_{m-1}1,\ nextUp(x) = 0 \\
& -2^{e_{min}} \cdot 0.\underbrace{0...0}_{m-1}1 < -2^{e_{min}} \cdot 0.m_y < 0 \quad \mid \cdot \tfrac{1}{2^{e_{min}}} \\
\Leftrightarrow\ & -0.\underbrace{0...0}_{m-1} < -0.m_y < 0 \mid \cdot (-1) \\
\Rightarrow\ & 0.\underbrace{0...0}_{m-1} > 0.m_y > 0 \mid << m \\
\Rightarrow\ & 1 > m_y > 0,\ m_y \in \mathbb{N}_{binary} \qquad\qquad\qquad\qquad \lightning \textit{Case } 4.2
\end{aligned}
$$

Case 5: $(x < 0) \wedge (x \text{ normal})$

Case 5.1: $x = -2^{e_x - \beta} \cdot 1.\underbrace{0...0}_{m}$

$$
\begin{aligned}
& -2^{e_x-\beta} \cdot 1.0 < -2^{e_y-\beta} \cdot 1.m_y < -2^{e_x-\beta-1} \cdot 1.\underbrace{1...1}_{m} \quad \mid y \textit{ normal} \Rightarrow e_y = e_x - 1 \\
\Rightarrow\ & -2^{e_x-\beta} \cdot 1.0 < -2^{e_x-\beta} \cdot 0.1m_y < -2^{e_x-\beta} \cdot 0.\underbrace{1...1}_{m+1} \quad \mid \cdot \tfrac{1}{2^{e_x-\beta}} \\
\Leftrightarrow\ & -1.0 < -0.1m_y < -0.\underbrace{1...1}_{m+1} \qquad\qquad \mid \cdot (-1);\ << (m+1) \\
\Leftrightarrow\ & 1\underbrace{0...0}_{m+1} > 1m_y > \underbrace{1...1}_{m+1},\ m_y \in \mathbb{N}_{binary} \qquad\qquad \lightning \textit{Case } 5.1
\end{aligned}
$$

Case 5.2: $x = -2^{e_x-\beta} \cdot 1.m_x,\ m_x \neq 0$

$$
\begin{aligned}
& -2^{e_x-\beta} \cdot 1.m_x < -2^{e_y-\beta} \cdot 1.m_y < -2^{e_x-\beta} \cdot 1.m_{nextUp(x)},\ m_{nextUp(x)} = m_x - 1 \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid normalized \Rightarrow e_y = e_x \\
\Rightarrow\ & -2^{e_x-\beta} \cdot 1.m_x < -2^{e_x-\beta} \cdot 1.m_y < -2^{e_x-\beta} \cdot 1.m_{nextUp(x)} \quad \mid \cdot \tfrac{1}{2^{e_x-\beta}} \\
\Leftrightarrow\ & -1.m_x < -1.m_y < 1.m_{nextUp(x)} \qquad\qquad \mid +1;\ \cdot(-1);\ << m \\
\Leftrightarrow\ & m_x > m_y > m_x - 1,\ m_y \in \mathbb{N}_{binary} \qquad\qquad \lightning \textit{Case } 5.2
\end{aligned}
$$

□

5.1.5 nextDown

The operation *nextDown* is defined by using *nextUp* as it is specified in [11]. *nextUp* is verified in Section 5.1.4, so *nextDown* is proven to be correct, too.

5.1.6 remainder

The operation *remainder* is defined as it is described in the IEEE Standard: $r = x - y \cdot n$ with n being the integral nearest to $\frac{x}{y}$, halfway cases can be arbitrarily rounded according to the IEEE Standard. The used rounding method is $roundToIntegralTiesToEven$, which rounds all floating point numbers to the next integral floating point number and halfway cases to the next even integral floating point number. That is conform to the IEEE description. And the fact that $fusedMultiplyAdd$ is used to calculate r eliminates the possibility of rounding errors during the calculation because nothing is rounded in between. That is why the definition of the operation *remainder* is known to be correct.

5.1.7 minNum

The operation $minNum$ is defined by using *compareQuietLess*, which is verified in Section 5.5.15, so $minNum$ is proven to be correct, too.

5.1.8 maxNum

The operation $maxNum$ is defined by using *compareQuietGreater*, which is verified in Section 5.5.13, so $maxNum$ is proven to be correct, too.

5.1.9 minNumMag

The operation $minNumMag$ is defined by using *compareQuietLess* and *abs*, which are verified in Section 5.5.15 and Section 5.4.3, so $minNum$ is proven to be correct, too.

5.1.10 maxNumMag

The operation $maxNumMag$ is defined by using *compareQuietGreater* and *abs*, which are verified in Section 5.5.13 and Section 5.4.3, so $maxNum$ is proven to be correct, too.

5.2 LogBFormat operations

5.2.1 scaleB

Theorem 2. *Let x be a floating point number, n be an integer and scaleB(x, n) the output of the operation defined in Figure 4.10. If no special cases occur the following holds*

$$scaleB(x,n) = \begin{cases} (-1)^{s_x} \cdot 2^{e_x - \beta + n} \cdot 1.m_x & \text{if } x \text{ is normal} \\ (-1)^{s_x} \cdot 2^{e_x + n + e'_r} \cdot (m_x << (-e'_r)) & \text{if } x \text{ is subnormal} \end{cases}$$

Proof. To prove Theorem 2 two cases have to be proven to be correct:

Case 1: x normal

$$\begin{aligned} &(-1)^{s_x} \cdot 2^{e_x - \beta} \cdot 1.m_x \cdot 2^n \\ &= (-1)^{s_x} \cdot 2^{e_x - \beta} \cdot 2^n \cdot 1.m_x \\ &= (-1)^{s_x} \cdot 2^{e_x - \beta + n} \cdot 1.m_x \end{aligned}$$

Case 2: x subnormal
$(-1)^{s_x} \cdot 2^{e_{min}} \cdot 0.m_x \cdot 2^n$
$= (-1)^{s_x} \cdot 2^{e_{min}} \cdot 2^n \cdot 2^{e'_r} \cdot (m_x << (-e'_r))$
$= (-1)^{s_y} \cdot 2^{e_{min}+n+e'_r} \cdot (m_x << (-e'_r))$s

□

5.2.2 logB

Theorem 3. *Let x be a floating point number and logB(x) the output of the operation defined in Figure 4.11. If no special cases occur the following holds*

$$logB(x) = \begin{cases} (e_x - \beta) + log_2(1.m_0) & \text{if } x \text{ normal} \\ e_{min} + log_2(0.m_0) & else \end{cases}$$

Proof. To prove Theorem 3 two cases have to be examined:

Case 1: x normal
$logB(x) = log_2(2^{e_x - \beta} \cdot 1.m_x)$
$= log_2(2^{e_x - \beta}) + log_2(1.m_x)$
$= (e_x - \beta) + log_2(1.m_x)$

Case 2: x subnormal
$logB(x) = log_2(2^{e_{min}} \cdot 0.m_x)$
$= log_2(2^{e_{min}}) + log_2(0.m_x)$
$= e_{min} + log_2(0.m_x)$

$logB(x)$ is equal to $log_2(x)$ because this arithmetic handles with binary numbers, so B is equal to 2. During the normalization, the mantissa is the only significant factor. Because of this, the calling of the lookup table that contains the log_2 for all mantissas returns the normalized resulting mantissa and the normalization factor for the resulting exponent. So the correctness of the result can be guaranteed. □

5.3 Arithmetic operations

5.3.1 addition

In addition to the following proof, the operation *addition* is completely tested for floating point numbers with bitlengths of eight bit and nine bit shown in Table 6.1.

To prove the correctness of the operation *addition* defined in Figure 4.12, two seperate proofs are performed.

Proof 1: $x + y = (-1)^{s_x} \cdot ||x| + (-1)^{s_x \oplus s_y} \cdot |y||$ with s_x and s_y according to the floating point format

Case 1: $x \leq 0,\ y \leq 0 \ \Rightarrow\ s_x = s_y = 1$

$x + y = -|x| + (-|y|) = -|x| + (-1) \cdot |y| = -(|x| - (-1) \cdot |y|) = (-1) \cdot (|x| + |y|)$

$= (-1) \cdot ||x| + |y|| = (-1)^1 \cdot ||x| + (-1)^0 \cdot |y|| = (-1)^{s_x} \cdot ||x| + (-1)^{s_x \oplus s_y} \cdot |y||$

$\square_{Case\ 1}$

Case 2: $x \leq 0,\ y \geq 0 \ \Rightarrow\ s_x = 1,\ s_y = 0$

$x + y = -|x| + |y| = -(|x| - \underbrace{|y|}_{\leq |x|}) = -||x| - |y|| = (-1)^1 \cdot ||x| + (-1)^1 \cdot |y||$

$\underbrace{\qquad\qquad}_{\geq 0}$

$= (-1)^{s_x} \cdot ||x| + (-1)^{s_x \oplus s_y} \cdot |y||$

$\square_{Case\ 2}$

Case 3: $x \geq 0,\ y \leq 0 \ \Rightarrow\ s_x = 0,\ s_y = 1$

$x + y = \underbrace{|x| - \underbrace{|y|}_{\leq |x|}}_{\geq 0} = ||x| - |y|| = (-1)^0 \cdot ||x| + (-1)^1 \cdot |y||$

$= (-1)^{s_x} \cdot ||x| + (-1)^{s_x \oplus s_y} \cdot |y||$

$\square_{Case\ 3}$

Case 4: $x \geq 0,\ y \geq 0 \ \Rightarrow\ s_x = s_y = 0$

$x + y = |x| + |y| = ||x| + |y|| = (-1)^0 \cdot ||x| + (-1)^0 \cdot |y||$

$= (-1)^{s_x} \cdot ||x| + (-1)^{s_x \oplus s_y} \cdot |y||$

$\square_{Case\ 4}$

$\square_{Proof\ 1}$

Proof 2: What remains to be proven is that no bits are required during the normalization that are erased before, i. e. G, R and S are sufficient.

Case 1: shift right

The three additional bits G, R and S are not required for shifting to the right. Basically, there is too much accuracy and bits are discarded. So shifting right cannot lead to a problem.

Case 2: shift left

Case 2.1: $S == 0$

This case cannot lead to a problem either, because all bits of the second operand are known.

Case 2.2: $S \neq 0$

The second operand was shifted by at least three bits, so the following holds if $addition(x, y)$ ($|x| \geq |y|$ because of swapping) holds after adjusting the mantissas according to the exponents:

$m'_y = 000 m_{y_{m-4}} ... m_{y_0} \leq 000 \underbrace{1...1}_{m-3}$ and $m_x = 1 m_{x_{m-2}} ... m_{x_0}$

It can be seen, that the second most significand bit $m_{r_{m-2}}$ of the resulting

mantissa is 1 and thus the maximal shifting amount before the rounding takes place is 1.

$\square_{Proof\ 2}$

$\square$

5.3.2 subtraction

The operation *subtraction* is defined by using *addition* which is verified in Section 5.3.1, so *subtraction* is proven to be correct, too.

5.3.3 multiplication

Theorem 4. *Let x and y be floating point numbers and multiplication(x, y) the output of the operation defined in Figure 4.15. If no special cases occur the following holds*

$$multiplication(x,y) = \begin{cases} (-1)^{s_x \oplus s_y} \cdot 2^{e_x + e_y - 2\cdot\beta} \cdot 1.m_x \cdot 1.m_y & \text{if } x \text{ normal, } y \text{ normal} \\ (-1)^{s_x \oplus s_y} \cdot 2^{e_x + e_{min} - \beta} \cdot 1.m_x \cdot 0.m_y & \text{if } x \text{ normal, } y \text{ subnormal} \\ (-1)^{s_x \oplus s_y} \cdot 2^{e_{min} + e_y - \beta} \cdot 0.m_x \cdot 1.m_y & \text{if } x \text{ subnormal, } y \text{ normal} \\ (-1)^{s_x \oplus s_y} \cdot 2^{2\cdot e_{min}} \cdot 0.m_x \cdot 0.m_y & \text{else} \end{cases}$$

Proof. To prove Theorem 4, four cases have to be examined:

Case 1: x normal, y normal

$x \cdot y = (-1)^{s_x} \cdot 2^{e_x - \beta} \cdot 1.m_x \cdot (-1)^{s_y} \cdot 2^{e_y - \beta} \cdot 1.m_y$
$= (-1)^{s_x + s_y} \cdot 2^{e_x - \beta + e_y - \beta} \cdot 1.m_x \cdot 1.m_y$
$= (-1)^{s_x + s_y} \cdot 2^{e_x + e_y - 2\cdot\beta} \cdot 1.m_x \cdot 1.m_y$
$= (-1)^{s_x \oplus s_y} \cdot 2^{e_x + e_y - 2\cdot\beta} \cdot 1.m_x \cdot 1.m_y$

$s_x s_y$	$(-1)^{s_x + s_y}$	$(-1)^{s_x \oplus s_y}$
00	$(-1)^0 = 1$	$(-1)^0 = 1$
01	$(-1)^1 = -1$	$(-1)^1 = -1$
10	$(-1)^1 = -1$	$(-1)^1 = -1$
11	$(-1)^2 = 1$	$(-1)^0 = 1$

Case 2: x normal, y subnormal

$x \cdot y = (-1)^{s_x} \cdot 2^{e_x - \beta} \cdot 1.m_x \cdot (-1)^{s_y} \cdot 2^{e_{min}} \cdot 0.m_y$
$= (-1)^{s_x + s_y} \cdot 2^{e_x - \beta + e_{min}} \cdot 1.m_x \cdot 0.m_y$
$= (-1)^{s_x + s_y} \cdot 2^{e_x + e_{min} - \beta} \cdot 1.m_x \cdot 0.m_y$
$= (-1)^{s_x \oplus s_y} \cdot 2^{e_x + e_{min} - \beta} \cdot 1.m_x \cdot 0.m_y$

Case 3: x subnormal, y normal

$x \cdot y = (-1)^{s_x} \cdot 2^{e_{min}} \cdot 0.m_x \cdot (-1)^{s_y} \cdot 2^{e_y - \beta} \cdot 1.m_y$
$= (-1)^{s_x + s_y} \cdot 2^{e_{min} + e_y - \beta} \cdot 0.m_x \cdot 1.m_y$
$= (-1)^{s_x + s_y} \cdot 2^{e_{min} + e_y - \beta} \cdot 0.m_x \cdot 1.m_y$
$= (-1)^{s_x \oplus s_y} \cdot 2^{e_{min} + e_y - \beta} \cdot 0.m_x \cdot 1.m_y$

Case 4: x subnormal, y subnormal

$$x \cdot y = (-1)^{s_x} \cdot 2^{e_{min}} \cdot 0.m_x \cdot (-1)^{s_y} \cdot 2^{e_{min}} \cdot 0.m_y$$
$$= (-1)^{s_x+s_y} \cdot 2^{e_{min}+e_{min}} \cdot 0.m_x \cdot 0.m_y$$
$$= (-1)^{s_x \oplus s_y} \cdot 2^{2 \cdot e_{min}} \cdot 0.m_x \cdot 0.m_y$$

□

5.3.4 division

Theorem 5. *Let x and y be floating point numbers and division(x, y) the output of the operation defined in Figure 4.16. If no special cases occur the following holds*

$$division(x,y) = \begin{cases} (-1)^{s_x \oplus s_y} \cdot 2^{e_x - e_y} \cdot \frac{1.m_x}{1.m_y} & \text{if } x \text{ normal, } y \text{ normal} \\ (-1)^{s_x \oplus s_y} \cdot 2^{e_x - \beta - e_{min}} \cdot \frac{1.m_x}{0.m_y} & \text{if } x \text{ normal, } y \text{ subnormal} \\ (-1)^{s_x \oplus s_y} \cdot 2^{e_{min} - e_y + \beta} \cdot \frac{0.m_x}{1.m_y} & \text{if } x \text{ subnormal, } y \text{ normal} \\ (-1)^{s_x \oplus s_y} \cdot \frac{0.m_x}{0.m_y} & \text{else} \end{cases}$$

Proof. To prove Theorem 5, four cases have to be examined:

Case 1: x normal, y normal

$$\frac{(-1)^{s_x} \cdot 2^{e_x-\beta} \cdot 1.m_x}{(-1)^{s_y} \cdot 2^{e_y-\beta} \cdot 1.m_y} = \frac{(-1)^{s_x}}{(-1)^{s_y}} \cdot \frac{2^{e_x-\beta}}{2^{e_y-\beta}} \cdot \frac{1.m_x}{1.m_y}$$
$$= (-1)^{s_x-s_y} \cdot 2^{e_x-\beta-(e_y-\beta)} \cdot \frac{1.m_x}{1.m_y}$$
$$= (-1)^{s_x \oplus s_y} \cdot 2^{e_x-e_y} \cdot \frac{1.m_x}{1.m_y}$$

$s_x s_y$	$(-1)^{s_x-s_y}$	$(-1)^{s_x \oplus s_y}$
00	$(-1)^0 = 1$	$(-1)^0 = 1$
01	$(-1)^-1 = -1$	$(-1)^1 = -1$
10	$(-1)^1 = -1$	$(-1)^1 = -1$
11	$(-1)^0 = 1$	$(-1)^0 = 1$

Case 2: x normal, y subnormal

$$\frac{(-1)^{s_x} \cdot 2^{e_x-\beta} \cdot 1.m_x}{(-1)^{s_y} \cdot 2^{e_{min}} \cdot 0.m_y} = \frac{(-1)^{s_x}}{(-1)^{s_y}} \cdot \frac{2^{e_x-\beta}}{2^{e_{min}}} \cdot \frac{1.m_x}{0.m_y}$$
$$= (-1)^{s_x-s_y} \cdot 2^{e_x-\beta-e_{min}} \cdot \frac{1.m_x}{0.m_y}$$
$$= (-1)^{s_x \oplus s_y} \cdot 2^{e_x-\beta-e_{min}} \cdot \frac{1.m_x}{0.m_y}$$

Case 3: x subnormal, y normal

$$\frac{(-1)^{s_x} \cdot 2^{e_{min}} \cdot 0.m_x}{(-1)^{s_y} \cdot 2^{e_y-\beta} \cdot 1.m_y} = \frac{(-1)^{s_x}}{(-1)^{s_y}} \cdot \frac{2^{e_{min}}}{2^{e_y-\beta}} \cdot \frac{0.m_x}{1.m_y}$$
$$= (-1)^{s_x-s_y} \cdot 2^{e_{min}-(e_y-\beta)} \cdot \frac{0.m_x}{1.m_y}$$
$$= (-1)^{s_x \oplus s_y} \cdot 2^{e_{min}-e_y+\beta} \cdot \frac{0.m_x}{1.m_y}$$

Case 4: x subnormal, y subnormal

$$\frac{(-1)^{s_x} \cdot 2^{e_{min}} \cdot 0.m_x}{(-1)^{s_y} \cdot 2^{e_{min}} \cdot 0.m_y} = \frac{(-1)^{s_x}}{(-1)^{s_y}} \cdot \frac{2^{e_{min}}}{2^{e_{min}}} \cdot \frac{0.m_x}{0.m_y}$$
$$= (-1)^{s_x-s_y} \cdot 1 \cdot \frac{0.m_x}{0.m_y}$$
$$= (-1)^{s_x \oplus s_y} \cdot \frac{0.m_x}{0.m_y}$$

□

5.3.5 squareRoot

Theorem 6. *Let x be a floating point number and squareRoot(x) the output of the operation defined in Figure 4.17. If no special cases occur the following holds*

$$squareRoot(x,y) = \begin{cases} 2^{(e_{min}-m_s)\ div\ 2} \cdot \sqrt{1.m_{x_{m-1-m_s}}...m_{x_0}\underbrace{0...0}_{m_s}} & \text{if } x \text{ subnormal}, \\ & (e_{min}-m_s)\ even \\ 2^{(e_{min}-m_s-1)\ div\ 2} \cdot \sqrt{1m_{x_{m-1-m_s}}.m_{x_{m-2-m_s}}...m_{x_0}\underbrace{0...0}_{m_s+1}} & \text{if } x \text{ subnormal}, \\ & (e_{min}-m_s)\ odd \\ 2^{(e_x-\beta)\ div\ 2} \cdot \sqrt{1.m_x} & \text{if } x \text{ normal}, \\ & (e_0-\beta)\ even \\ 2^{(e_x-\beta-1)\ div\ 2} \cdot \sqrt{1m_{x_{m-1}}.m_{x_{m-2}}...m_{x_0}0} & else \end{cases}$$

Proof. To prove Theorem 6, four cases have to be examined:

Case 1: x subnormal, $(e_{min}-m_s)$ even

$$\begin{aligned} \sqrt{2^{e_{min}} \cdot 0.m_x} &= \sqrt{2^{e_{min}-m_s} \cdot 1.m_{x_{m-1-m_s}}...m_{x_0}\underbrace{0...0}_{m_s}} \\ &= \sqrt{2^{e_{min}-m_s}} \cdot \sqrt{1.m_{x_{m-1-m_s}}...m_{x_0}\underbrace{0...0}_{m_s}} = 2^{\frac{e_{min}-m_s}{2}} \cdot \sqrt{1.m_{x_{m-1-m_s}}...m_{x_0}\underbrace{0...0}_{m_s}} \\ &= 2^{(e_{min}-m_s)\ div\ 2} \cdot \sqrt{1.m_{x_{m-1-m_s}}...m_{x_0}\underbrace{0...0}_{m_s}} \end{aligned}$$

Case 2: x subnormal, $(e_{min}-m_s)$ odd

$$\begin{aligned} \sqrt{2^{e_{min}} \cdot 0.m_x} &= \sqrt{2^{e_{min}-m_s-1} \cdot 1m_{x_{m-1-m_s}}.m_{x_{m-2-m_s}}...m_{x_0}\underbrace{0...0}_{m_s+1}} \\ &= \sqrt{2^{e_{min}-m_s-1}} \cdot \sqrt{1m_{x_{m-1-m_s}}.m_{x_{m-2-m_s}}...m_{x_0}\underbrace{0...0}_{m_s+1}} \\ &= 2^{\frac{e_{min}-m_s-1}{2}} \cdot \sqrt{1m_{x_{m-1-m_s}}.m_{x_{m-2-m_s}}...m_{x_0}\underbrace{0...0}_{m_s+1}} \\ &= 2^{(e_{min}-m_s-1)\ div\ 2} \cdot \sqrt{1m_{x_{m-1-m_s}}.m_{x_{m-2-m_s}}...m_{x_0}\underbrace{0...0}_{m_s+1}} \end{aligned}$$

Case 3: x normal, $(e_x-\beta-m_s)$ even

$$\sqrt{2^{e_x-\beta} \cdot 1.m_x} = \sqrt{2^{e_x-\beta}} \cdot \sqrt{1.m_x} = 2^{\frac{e_x-\beta}{2}} \cdot \sqrt{1.m_x} = 2^{(e_x-\beta)\ div\ 2} \cdot \sqrt{1.m_x}$$

Case 4: x normal, $(e_x-\beta-m_s)$ odd

$$\begin{aligned} \sqrt{2^{e_x-\beta} \cdot 1.m_x} &= \sqrt{2^{e_x-\beta-1} \cdot 1m_{x_{m-1}}.m_{x_{m-2}}...m_{x_0}0} = \sqrt{2^{e_x-\beta-1}} \cdot \sqrt{1m_{x_{m-1}}.m_{x_{m-2}}...m_{x_0}0} \\ &= 2^{\frac{e_x-\beta-1}{2}} \cdot \sqrt{1m_{x_{m-1}}.m_{x_{m-2}}...m_{x_0}0} = 2^{(e_x-\beta-1)\ div\ 2} \cdot \sqrt{1m_{x_{m-1}}.m_{x_{m-2}}...m_{x_0}0} \end{aligned}$$

Remains to be proven that $\sqrt{1x_n.x_{n-1}...x_0} \geq 1.0$ and $\sqrt{1.x_n...x_0} \geq 1.0$ holds:

Case 1: $x := 1x_n.x_{n-1}...x_0 \geq 1.0$

$$\begin{aligned} & x \geq 1.0 && | \sqrt{}\ ;\ x = 1x_n.x_{n-1}...x_0 \geq 0 \\ \Rightarrow\ & \sqrt{x} \geq \sqrt{1.0} && | \ 1.0 = 1.0^2 \\ \Rightarrow\ & \sqrt{x} \geq \sqrt{1.0^2} \\ \Rightarrow\ & \sqrt{x} \geq 1.0 \end{aligned}$$

Case 2: $x := 1.x_n...x_0 \geq 1.0$

$$\begin{aligned} & x \geq 1.0 && | \sqrt{}\ ;\ x = 1.x_n...x_0 \geq 0 \\ \Rightarrow\ & \sqrt{x} \geq \sqrt{1.0} && | \ 1.0 = 1.0^2 \\ \Rightarrow\ & \sqrt{x} \geq \sqrt{1.0^2} \\ \Rightarrow\ & \sqrt{x} \geq 1.0 \end{aligned}$$

Thus, the normalization cannot fail due to a lack of accuracy. The case that the resulting mantissa has to be shifted left cannot occur, because it is always greater or equal to one. And shifting to the right reduces the accuracy of the mantissa, so no additional bits are required. □

The calculation of the square root is a fixed point operation. The high/low method [8, 25] provides the possibility to calculate an approximation of the square root with a given accuracy, and this is perfectly suitable for calculating the resulting mantissa.

5.3.6 fusedMultiplyAdd

The operation $fusedMultiplyAdd$ defined in Figure 4.18 uses the operations *multiplication* and *addition*. These operations are verified in Section 5.3.3 and Section 5.3.1 and the calculations are done with double precision, so the operation $fusedMultiplyAdd$ is verified implicitly.

5.3.7 convertFromInt

The operation $convertFromInt$ returns the floating point representation of a given integer. That conversion is completely defined in the IEEE Standard 754 and the operation defined in this thesis is conform to it. Because of that, the definition has to be correct and a formal verification is dispensible.

5.3.8 convertToInteger*

The operations $convertToIntegerTiesToEven$ and $convertToIntegerExactTiesToEven$ convert a given floating point number to the nearest integer while halfway cases are rounded to the nearest even integer. If $convertToIntegerExactTiesToEven$ and the input was not an integral floating point value, an exception is raised. That are exactly the requirements defined in the IEEE Standard, therefore the definition is conform to the IEEE Standard 754-2008 and a formal verification is not required. The same applies to the operations $convertToIntegerTiesToAway$ and $convertToIntegerExactTiesToAway$, that convert a given floating point number to the nearest integer while halfway cases are rounded to the nearest integer away from zero.
The following operations are always rounded in the same direction if the input does not

have an integral floating point value. The operations $convertToIntegerTowardZero$ and $convertToIntegerExactTowardZero$ are always rounded towards zero, while the operations $convertToIntegerTowardPositive$ and $convertToIntegerExactTowardPositive$ are always rounded towards $+\infty$ and analogous, the operations $convertToIntegerTowardNegative$ and $convertToIntegerExactTowardNegative$ are always rounded towards $-\infty$. Again, that are exactly the requirements defined in the IEEE Standard, what supersedes a formal verification.

5.4 Sign bit operations

5.4.1 copy

The operation *copy* forwards the input to the output without any changes. So there is nothing that has to be verified.

5.4.2 negate

Theorem 7. *Let x be a floating point number and negate(x) the output of the operation defined in Figure 4.25 with x as input. If no special cases occur the following holds*

$$\forall x \in FP : (-x) == [\neg s_x, e_x, m_x]$$

Proof. $-x = (-1) \cdot (-1)^{s_x} \cdot 2^{e_x-\beta} \cdot m_x = (-1)^{s_x+1} \cdot 2^{e_x-\beta} \cdot m_x$
$= (-1)^{(s_x+1) \ mod \ 2} \cdot 2^{e_x-\beta} \cdot m_x = (-1)^{\neg s_x} \cdot 2^{e_x-\beta} \cdot m_x = [\neg s_x, e_x, m_x]$ □

5.4.3 abs

The definition of the operation *abs* shown in Figure 4.26 returns the input value for positive inputs and uses the operation *negate*. This operation is verified in Section 5.4.2, so the operation *abs* is verified implicitly.

5.4.4 copySign

The operation *copySign* forwards the sign of the first input and exponent and mantissa of the second input to the output without any changes. So there is nothing that has to be verified.

5.5 Comparison operations

5.5.1 compareQuietEqual

Theorem 8. *Let x, y be floating point numbers and compareQuietEqual(x,y) the output of the operation defined in Figure 4.28 with x and y as inputs. If no special cases occur the following holds*

$$\forall x \in FP \nexists y \in FP : (x == y) \wedge ((s_x \neq s_y) \vee (e_x \neq e_y) \vee (m_x \neq m_y)) \wedge (x \neq 0).$$

Proof. To prove Theorem 8, reductio ad absurdum is used. The claim

$$\exists x, y \in FP : (x == y) \wedge ((s_x \neq s_y) \vee (e_x \neq e_y) \vee (m_x \neq m_y)) \wedge (x \neq 0)$$

is proven to be wrong.

Case 1: $(x == y) \wedge (s_x \neq s_y) \wedge (x \neq 0)$

$\Leftrightarrow \quad (x == y) \wedge ((x < 0) \wedge (y > 0) \vee (x > 0) \wedge (y < 0)) \wedge (x \neq 0) \quad | y = x$

$\Leftrightarrow \quad ((x < 0) \wedge (x > 0) \vee (x > 0) \wedge (x < 0)) \wedge (x \neq 0)$ $\quad \lightning \mathit{Case}\ 1$

Case 2: $(x == y) \wedge (e_x \neq e_y) \wedge (x \neq 0)$

Case 2.1: x subnormal, y normal

$e_x == 0 \quad \Rightarrow$ h.b.: $0 \quad \Rightarrow \; x = (-1)^{s_x} \cdot 2^{e_{min}} \cdot 0.m_x$

$e_y \neq 0 \quad \Rightarrow$ h.b.: $1 \quad \Rightarrow \; y = (-1)^{s_y} \cdot 2^{e_y - \beta} \cdot 1.m_y$

$((-1)^{s_x} \cdot 2^{e_{min}} \cdot 0.m_x == (-1)^{s_y} \cdot 2^{e_y - \beta} \cdot 1.m_y) \wedge$

$(e_y \neq 0) \wedge (m_x \neq 0) \quad | \mathit{Case}\ 1 \Rightarrow s_x = s_y$

$\Rightarrow \quad (2^{e_{min}} \cdot 0.m_x == 2^{e_y - \beta} \cdot 1.m_y) \wedge (e_y \neq 0) \wedge (m_x \neq 0)$

$\Rightarrow \quad e_y < e_{min} + \beta$

$\Rightarrow \quad e_y < 1 - \beta + \beta$

$\Rightarrow \quad e_y < 1 \quad | e_y \in \mathbb{N}$

$\Rightarrow \quad e_y \leq 0$ $\quad \lightning \mathit{Case}\ 2.1$

Case 2.2: x normal, y normal

$e_x > 0 \quad \Rightarrow$ h.b.: $1 \quad \Rightarrow \; x = (-1)^{s_x} \cdot 2^{e_x - \beta} \cdot 1.m_x$

$e_y > 0 \quad \Rightarrow$ h.b.: $1 \quad \Rightarrow \; y = (-1)^{s_y} \cdot 2^{e_y - \beta} \cdot 1.m_y$

$e_x \neq e_y \quad \Rightarrow \; (e_x > e_y) \vee (e_x < e_y)$

$(x == y) \wedge ((e_x > e_y) \vee (e_x < e_y)) \wedge (e_x \neq 0) \wedge (e_y \neq 0)$

$\Rightarrow \quad (x == y) \wedge ((\text{h.b.}(y) == 1 << (e_x - e_y)) \vee$

$(\text{h.b.}(x) == 1 << (e_y - e_x))) \wedge (e_x \neq 0) \wedge (e_y \neq 0)$

x and y are normalized $\quad \lightning \mathit{Case}\ 2.2$

Case 2.3: x subnormal, y normal

see Case 2.1

Case 2.4: x subnormal, y subnormal

$\lightning e_x \neq e_y$

Case 3: $(x == y) \wedge (m_x \neq m_y) \wedge (x \neq 0)$

$(((-1)^{s_x} \cdot 2^{e_x - \beta} \cdot 1.m_x == (-1)^{s_y} \cdot 2^{e_y - \beta} \cdot 1.m_y) \vee$

$((-1)^{s_x} \cdot 2^{e_x - \beta} \cdot 1.m_x == (-1)^{s_y} \cdot 2^{e_y - \beta} \cdot 0.m_y) \vee$

$((-1)^{s_x} \cdot 2^{e_x - \beta} \cdot 0.m_x == (-1)^{s_y} \cdot 2^{e_y - \beta} \cdot 1.m_y) \vee$ $\quad | \mathit{Case}1, \mathit{Case}2 \Rightarrow s_y = s_x, e_y = e_x$

$((-1)^{s_x} \cdot 2^{e_x - \beta} \cdot 0.m_x == (-1)^{s_y} \cdot 2^{e_y - \beta} \cdot 0.m_y)) \wedge$

$(m_x \neq m_y) \wedge (x \neq 0)$

$\Rightarrow \quad ((1.m_x == 1.my) \vee (1.m_x == 0.m_y) \vee$

$(0.m_x == 1.m_y) \vee (0.m_x == 0.m_y)) \wedge$

$(m_x \neq m_y) \wedge (x \neq 0)$ $\quad \lightning \mathit{Case}\ 3$

□

5.5.2 compareQuietNotEqual

The definition of the operation *compareQuietNotEqual* shown in Figure 4.29 uses the operation *compareQuietEqual*. This operation is verified in Section 5.5.1, so the operation *compareQuietNotEqual* is verified implicitly.

5.5.3 compareSignalingEqual

The definition of the operation *compareSignalingEqual* shown in Figure 4.30 uses the operation *compareQuietEqual*. This operation is verified in Section 5.5.1, so the operation *compareSignalingEqual* is verified implicitly.

5.5.4 compareSignalingGreater

The definition of the operation *compareSignalingGreater* shown in Figure 4.31 uses the operation *compareQuietGreater*. This operation is verified in Section 5.5.13, so the operation *compareSignalingGreater* is verified implicitly.

5.5.5 compareSignalingGreaterEqual

The definition of the operation *compareSignalingGreaterEqual* shown in Figure 4.32 uses the operation *compareQuietGreaterEqual*. This operation is verified in Section 5.5.14, so the operation *compareSignalingGreaterEqual* is verified implicitly.

5.5.6 compareSignalingLess

The definition of the operation *compareSignalingLess* shown in Figure 4.33 uses the operation *compareQuietLess*. This operation is verified in Section 5.5.15, so the operation *compareSignalingLess* is verified implicitly.

5.5.7 compareSignalingLessEqual

The definition of the operation *compareSignalingLessEqual* shown in Figure 4.34 uses the operation *compareQuietLessEqual*. This operation is verified in Section 5.5.16, so the operation *compareSignalingLessEqual* is verified implicitly.

5.5.8 compareSignalingNotEqual

The definition of the operation *compareSignalingNotEqual* shown in Figure 4.35 uses the operation *compareQuietNotEqual*. This operation is verified in Section 5.5.2, so the operation *compareSignalingNotEqual* is verified implicitly.

5.5.9 compareSignalingNotGreater

The definition of the operation *compareSignalingNotGreater* shown in Figure 4.36 uses the operation *compareQuietNotGreater*. This operation is verified in Section 5.5.18, so the operation *compareSignalingNotGreater* is verified implicitly.

5.5.10 compareSignalingLessUnordered

The definition of the operation *compareSignalingLessUnordered* shown in Figure 4.37 uses the operation *compareQuietLess*. This operation is verified in Section 5.5.15, so the operation *compareSignalingLessUnordered* is verified implicitly.

5.5.11 compareSignalingNotLess

The definition of the operation *compareSignalingNotLess* shown in Figure 4.38 uses the operation *compareQuietNotLess*. This operation is verified in Section 5.5.20, so the operation *compareSignalingNotLess* is verified implicitly.

5.5.12 compareSignalingGreaterUnordered

The definition of the operation *compareSignalingGreaterUnordered* shown in Figure 4.39 uses the operation *compareQuietGreater*. This operation is verified in Section 5.5.13, so the operation *compareSignalingGreaterUnordered* is verified implicitly.

5.5.13 compareQuietGreater

Theorem 9. *Let x, y be floating point numbers and compareQuietGreater(x,y) the output of the operation defined in Figure 4.40 with x and y as inputs. If no special cases occur the following holds*

$$\forall x, y \in FP : ((s_x < s_y) \vee ((s_x == s_y) \wedge (e_x > e_y)) \vee ((s_x == s_y) \wedge (e_x == e_y) \wedge (m_x > m_y))) \wedge$$

$$(compareQuietGreater(x, y) == True) \vee$$

$$((s_x > s_y) \vee ((s_x == s_y) \wedge (e_x < e_y)) \vee ((s_x == s_y) \wedge (e_x == e_y) \wedge (m_x < m_y)) \vee$$

$$((s_x == s_y) \wedge (e_x == e_y) \wedge (m_x == m_y))) \wedge (compareQuietGreater(x, y) == False)$$

Proof. To prove Theorem 9 seven cases have to be examined

Case 1: $(x > y) \wedge (s_x > s_y)$
$= (x > y) \wedge (s_x > s_y)$
$= (x > y) \wedge (s_x == 1) \wedge (s_y == 0)$
$= (x > y) \wedge (x \leq 0) \wedge (y \geq 0)$ $\lightning_{Case\ 1}$

Case 2: $(x > y) \wedge (s_x < s_y)$
$= (x > y) \wedge (s_x == 0) \wedge (s_y == 1)$
$= (x > y) \wedge (x \geq 0) \wedge (y \leq 0)$
$(-0 == +0)$ covered in special cases $\square_{Case\ 2}$

Case 3: $(x > y) \wedge (s_x == s_y) \wedge (e_x > e_y)$

Case 3.1: $(s_x == 0)$
$(x > y) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (e_x > e_y)$
$=_{normalized} (x > y) \wedge (x > y)$ $\square_{Case\ 3.1}$

Case 3.2: $(s_x == 1)$

$(x > y) \wedge (x \leq 0) \wedge (y \leq 0) \wedge (e_x > e_y)$

$=_{normalized} (x > y) \wedge (-x > -y)$

$=_{normalized} (x > y) \wedge (x < y)$ $\lightning_{Case\ 3.2}$

Case 4: $(x > y) \wedge (s_x == s_y) \wedge (e_x < e_y)$

Case 4.1: $(s_x == 0)$

$(x > y) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (e_x < e_y)$

$=_{normalized} (x > y) \wedge (x < y)$ $\lightning_{Case\ 4.1}$

Case 4.2: $(s_x == 1)$

$(x > y) \wedge (x \leq 0) \wedge (y \leq 0) \wedge (e_x < e_y)$

$=_{normalized} (x > y) \wedge (-x < -y)$

$=_{normalized} (x > y) \wedge (x > y)$ $\square_{Case\ 4.2}$

Case 5: $(x > y) \wedge (s_x == s_y) \wedge (e_x == e_y) \wedge (m_x > m_y)$

Case 5.1: $(s_x == 0)$

$(x > y) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (e_x == e_y) \wedge (m_x > m_y)$

$= (x > y) \wedge (x > y)$ $\square_{Case\ 5.1}$

Case 5.2: $(s_x == 1)$

$(x > y) \wedge (x \leq 0) \wedge (y \leq 0) \wedge (e_x == e_y) \wedge (m_x > m_y)$

$= (x > y) \wedge (-x > -y)$

$= (x > y) \wedge (x < y)$ $\lightning_{Case\ 5.2}$

Case 6: $(x > y) \wedge (s_x == s_y) \wedge (e_x == e_y) \wedge (m_x < m_y)$

Case 6.1: $(s_x == 0)$

$(x > y) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (e_x == e_y) \wedge (m_x < m_y)$

$= (x > y) \wedge (x < y)$ $\lightning_{Case\ 6.1}$

Case 6.2: $(s_x == 1)$

$(x > y) \wedge (x \leq 0) \wedge (y \leq 0) \wedge (e_x == e_y) \wedge (m_x < m_y)$

$(x > y) \wedge (-x < -y)$

$(x > y) \wedge (x > y)$ $\square_{Case\ 6.2}$

Case 7: $(x > y) \wedge (s_x == s_y) \wedge (e_x == e_y) \wedge (m_x == m_y)$

$(x > y) \wedge (s_x == s_y) \wedge (e_x == e_y) \wedge (m_x == m_y)$

$= (x > y) \wedge (x == y)$ $\lightning_{Case\ 7}$

$\square$

5.5.14 compareQuietGreaterEqual

The definition of the operation *compareQuietGreaterEqual* shown in Figure 4.42 uses the operations *compareQuietEqual* and *compareQuietGreater*. These operations are verified in Section 5.5.1 and Section 5.5.13, so the operation *compareQuietGreaterEqual* is verified implicitly.

5.5.15 compareQuietLess

Theorem 10. *Let x, y be floating point numbers and compareQuietLess(x,y) the output of the operation defined in Figure 4.41 with x and y as inputs. If no special cases occur the following holds*

$$\forall x, y \in FP : ((s_x > s_y) \vee ((s_x == s_y) \wedge (e_x < e_y)) \vee ((s_x == s_y) \wedge (e_x == e_y) \wedge (m_x < m_y))) \wedge$$

$$(compareQuietLess(x, y) == True) \vee$$

$$((s_x < s_y) \vee ((s_x == s_y) \wedge (e_x > e_y)) \vee ((s_x == s_y) \wedge (e_x == e_y) \wedge (m_x > m_y)) \vee$$

$$((s_x == s_y) \wedge (e_x == e_y) \wedge (m_x == m_y))) \wedge (compareQuietLess(x, y) == False)$$

Proof. To prove Theorem 10 seven cases have to be examined

Case 1: $(x < y) \wedge (s_x > s_y)$
$= (x < y) \wedge (s_x > s_y)$
$= (x < y) \wedge (s_x == 1) \wedge (s_y == 0)$
$= (x < y) \wedge (x \leq 0) \wedge (y \geq 0)$
$(-0 == +0)$ covered in special cases $\square_{Case\ 1}$

Case 2: $(x < y) \wedge (s_x < s_y)$
$= (x < y) \wedge (s_x == 0) \wedge (s_y == 1)$
$= (x < y) \wedge (x \geq 0) \wedge (y \leq 0)$
$\Rightarrow (x < y) \wedge (x \geq y)$ $\lightning_{Case\ 2}$

Case 3: $(x < y) \wedge (s_x == s_y) \wedge (e_x > e_y)$

Case 3.1: $(s_x == 0)$
$(x < y) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (e_x > e_y)$
$=_{normalized} (x < y) \wedge (x > y)$ $\lightning_{Case\ 3.1}$

Case 3.2: $(s_x == 1)$
$(x < y) \wedge (x \leq 0) \wedge (y \leq 0) \wedge (e_x > e_y)$
$=_{normalized} (x < y) \wedge (-x > -y)$
$=_{normalized} (x < y) \wedge (x < y)$ $\square_{Case\ 3.2}$

Case 4: $(x < y) \wedge (s_x == s_y) \wedge (e_x < e_y)$

Case 4.1: $(s_x == 0)$
$(x < y) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (e_x < e_y)$
$=_{normalized} (x < y) \wedge (x < y)$ $\square_{Case\ 4.1}$

Case 4.2: $(s_x == 1)$
$(x < y) \wedge (x \leq 0) \wedge (y \leq 0) \wedge (e_x < e_y)$
$=_{normalized} (x < y) \wedge (-x < -y)$
$=_{normalized} (x < y) \wedge (x > y)$ $\lightning_{Case\ 4.2}$

Case 5: $(x < y) \wedge (s_x == s_y) \wedge (e_x == e_y) \wedge (m_x > m_y)$

Case 5.1: $(s_x == 0)$

$(x < y) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (e_x == e_y) \wedge (m_x > m_y)$

$= (x < y) \wedge (x > y)$ $\lightning_{Case\ 5.1}$

Case 5.2: $(s_x == 1)$

$(x < y) \wedge (x \leq 0) \wedge (y \leq 0) \wedge (e_x == e_y) \wedge (m_x > m_y)$

$= (x < y) \wedge (-x > -y)$

$= (x < y) \wedge (x < y)$ $\square_{Case\ 5.2}$

Case 6: $(x < y) \wedge (s_x == s_y) \wedge (e_x == e_y) \wedge (m_x < m_y)$

Case 6.1: $(s_x == 0)$

$(x < y) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (e_x == e_y) \wedge (m_x < m_y)$

$= (x < y) \wedge (x < y)$ $\square_{Case\ 6.1}$

Case 6.2: $(s_x == 1)$

$(x < y) \wedge (x \leq 0) \wedge (y \leq 0) \wedge (e_x == e_y) \wedge (m_x < m_y)$

$(x < y) \wedge (-x < -y)$

$(x < y) \wedge (x > y)$ $\lightning_{Case\ 6.2}$

Case 7: $(x < y) \wedge (s_x == s_y) \wedge (e_x == e_y) \wedge (m_x == m_y)$

$(x < y) \wedge (s_x == s_y) \wedge (e_x == e_y) \wedge (m_x == m_y)$

$= (x < y) \wedge (x == y)$ $\lightning_{Case\ 7}$

$\square$

5.5.16 compareQuietLessEqual

The definition of the operation *compareQuietLessEqual* shown in Figure 4.43 uses the operations *compareQuietEqual* and *compareQuietLess*. These operations are verified in Section 5.5.1 and Section 5.5.15, so the operation *compareQuietLessEqual* is verified implicitly.

5.5.17 compareQuietUnordered

The definition of the operation *compareQuietUnordered* shown in Figure 4.44 returns $True$ if at least one of the inputs is a NaN and returns $False$ otherwise. An exception is not raised according to the IEEE Standard. So the definition fulfills all requirements.

5.5.18 compareQuietNotGreater

The definition of the operation *compareQuietNotGreater* shown in Figure 4.45 uses the operation *compareQuietGreater*. This operation is verified in Section 5.5.13, so the operation *compareQuietGreaterEqual* is verified implicitly.

5.5.19 compareQuietLessUnordered

The definition of the operation *compareQuietLessUnordered* shown in Figure 4.46 uses the operation *compareQuietLess*. This operation is verified in Section 5.5.15, so the operation *compareQuietLessUnordered* is verified implicitly.

5.5.20 compareQuietNotLess

The definition of the operation *compareQuietNotLess* shown in Figure 4.47 uses the operation *compareQuietLess*. This operation is verified in Section 5.5.15, so the operation *compareQuietNotLess* is verified implicitly.

5.5.21 compareQuietGreaterUnordered

The definition of the operation *compareQuietGreaterUnordered* shown in Figure 4.48 uses the operation *compareQuietGreater*. This operation is verified in Section 5.5.13, so the operation *compareQuietGreaterUnordered* is verified implicitly.

5.5.22 compareQuietOrdered

The definition of the operation *compareQuietOrdered* shown in Figure 4.49 returns $False$ if at least one of the inputs is a NaN and returns $True$ otherwise. An exception is not raised according to the IEEE Standard. So the definition fulfills all requirements.

6 Implementation

The arithmetic operations defined in Chapter 4 are implemented using the synchronous programming language Quartz [23]. In the majority of cases, the implementation can more or less be transcribed from the definition.

One major advantage in Quartz is the support of bitvectors with arbitrary length, i.e. one does not have to map bitvectors to types that are natively supported, e.g. integers. Moreover, Quartz targets to create both, hardware and software systems.

But as a matter of course, some adaptions have to be made and some restrictions have to be considered.
Many operations defined in Section 4.2 make use of shifting operations. The original purpose of Quartz is to be used as a hardware description language. That is why it does not offer shifting operations with a dynamic shift-amount and operations using these have to be implemented in another way, e. g. by using a Barrel shifter [20].
Furthermore, variables are statically allocated, which is also reasoned by the targeted architectures of Quartz, i.e. hardware and software. Hardware is obviously not capable of dynamic changes.
Another issue is the fact that global variables used for the exceptions cannot be implemented. For that reason and because it does not change the value of the results of the functions, the raising of exceptions is neglected in this implementation. Also omited are the operations grouped in conversion operations, because Quartz does not provide strings and therefore the representation of hexadecimal outputs as well as hexadecimal inputs would be inconvenient.

Beside the shifting operations mentioned before a fixed point square root calculation is implemented, because the implementation of the floating point square root builds on it.

length of			
sign	exponent	mantissa	bitlength
1	3	4	8
1	2	5	8
1	3	5	9
1	4	5	10
1	8	23	single-precision (32-bit)

Table 6.1: Bitlengths Used for Test of Implementation

To check the correctness of the implementation and double-check the completeness and the validity of the definitions, the implementations are testes with several arrangements of lengths of mantissa and exponent. The tested bitlengths are depicted in Table 6.1.

7 Conclusion and Future Work

7.1 Conclusion

This thesis presents the definition, verification and implementation of an IEEE-conform floating point arithmetic for binary floating point numbers with arbitrary bitlengths.

The starting point was the IEEE Standard 754-2008 [11]. After a thorough review of this standard, the requirements on the definitions are gleaned. These requirements specify the format, mandatory operations, handling of special cases and exceptions. Based on this and using the publications listed in the bibliography, the definitions were elaborated.
After the arithmetic was defined, the operations were verified. If necessary, the verifications are written down formally. Simultaneous, the arithmetic was implemented using the synchronous programming language Quartz. These implementations were tested for multiple bitlengths.

Due to the focus on the correctness and to ease the verification, optimizations were not attempted during writing this thesis. But based on the provided arithmetic that is proven to be correct, new definitions and implementations can be tested for being IEEE-conform more easily.

7.2 Future Work

As mentioned in the conclusion in Section 7.1, the definitions as well as the implementations have the potential of being optimized, e. g. with respect to their runtime or their resource consumption.

Bibliography

[1] Herbert Bruderer. *Konrad Zuse und die Schweiz: Wer hat den Computer erfunden?* Oldenbourg Wissenschaftsverlag, 2012.

[2] W. Cody, J. Coonen, D. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. Ris, and D. Stevenson. A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic. *IEEE Micro*, 4(4):86–100, July 1984.

[3] Jerome Coonen, William Kahan, John Palmer, Tom Pittman, and David Stevenson. A Proposed Standard for Binary Floating Point Arthmetic. *SIGNUM Newsl.*, 14(si-2):4–12, October 1979.

[4] Harvey G. Cragon. *Computer Architecture and Implementation.* Cambridge University Press, 2000.

[5] Tilo Fischer and Hans J Dorn. *Pysikalische Formeln und Daten.* Klett, 1986.

[6] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[7] James Gosling. *The Java Language Specification, Second Edition.* Addison-Wesley Longman, Amsterdam, 2000.

[8] Prof. Dr.-Ing. habil. Alexander Potchinkov. Digitale Signalverarbeitung: Algorithmen und Implementierung. Presentation of Course, Chapter 2, April 2010.

[9] IBM. *Language Environment for OS/390 & VM Vendor Interfaces - IEEE Floating-Point Supplement.* International Business Machines Corporation, 1998.

[10] IEEE. *IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std 754-1985).* Institute of Electrical and Electronics Engineers, Inc., 1985.

[11] IEEE. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008).* Institute of Electrical and Electronics Engineers, Inc., August 2008.

[12] R. R. Rogers C. F. Webb K. E. Plambeck, E. Eckert. *Development and attributes o z/Architecture.* 2002.

[13] Prof. W. Kahan and Joseph D. Darcy. How Java's Floating-Point Hurts Everyone Everywhere. March 1998.

[14] W. Kahan and J. Palmer. On a Proposed Floating-point Standard. *SIGNUM Newsl.*, 14(si-2):13–21, October 1979.

[15] Schaeffler Technologies AG & Co. KG. *Technisches Taschenbuch*. 2013.

[16] Loucas Louca, Todd A. Cook, and William H. Johnson. Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs. *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 107–116, April 1996.

[17] International Business Machines. *IBM 704 Manual of Operation*. 1954.

[18] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serfe Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2009.

[19] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics Philadelphia, 2001.

[20] Matthew Rudolf Pillmeier. *Barrel shifter design, optimization, and analysis*. PhD thesis, Lehigh University, 2001. Paper 714.

[21] Prof. Dr. rer. nat. Klaus Schneider. Computer Systems 1 & 2, 2011. Lecture Notes.

[22] Raúl Rojas. *Die Architektur der Rechenmaschinen Z1 und Z3*. Springer Berlin Heidelberg, 1998.

[23] Klaus Schneider. *The Synchronous Programming Language Quartz*. Department of Computer Science University of Kaiserslautern, November 2010.

[24] Dr. Bernd Schürmann. Begleitschrift zu den Vorlesungen "Rechnersysteme 1 und 2", 2011. Lecture Notes.

[25] Jacqueline Stratmann. Design and Implementation of a Floating-Point Application Specific Instruction Set Processor for Asset Simulations. Bachelorthesis.

[26] Oriol Vinyals, Gerald Friedland, and Nikki Mirghafori. Revisiting a basic function on current CPUs: A fast logarithm implementation with adjustable accuracy. *International Computer Science Institute*, June 2007.