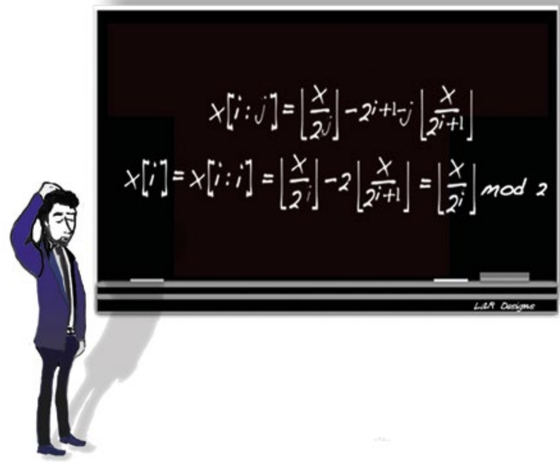
David M. Russinoff



Formal Verification of Floating-Point Hardware Design

A Mathematical Approach

Springer

Formal Verification of Floating-Point Hardware Design

David M. Russinoff

Formal Verification of Floating-Point Hardware Design

A Mathematical Approach

Foreword by J Strother Moore

David M. Russinoff
Arm Holdings
Austin, TX, USA

ISBN 978-3-319-95512-4 ISBN 978-3-319-95513-1 (eBook)
https://doi.org/10.1007/978-3-319-95513-1

Library of Congress Control Number: 2018949694

© Springer Nature Switzerland AG 2019
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Lin, Joshua, and Solomon

Foreword

Stone masons were building bridges—and using mathematics—long before 1773, when Coulomb published his groundbreaking mathematical analysis of some fundamental problems in civil engineering: the bending of beams, the failure of columns, and the determination of abutment thrusts imposed by arches. While Coulomb's work was appreciated by the mathematicians and physicists of the day, it was largely irrelevant to the bridge builders who were guided by experience, tradition, and intuition. However, over the next several centuries attitudes changed; statics is now a standard part of the civil engineer's training. One reason is that as materials and requirements changed, and as mathematics and computation further developed, what was once impractical became practical and then, perhaps, automatic. Mathematical tools facilitated the design of safe, cost-effective, reliable structures that could not have been built by earlier techniques.

But acceptance took time and, sometimes, dreadful experience. For example, a catastrophic bridge failure occurred a full century after Coulomb's work. The bridge carrying the Edinburgh to Dundee train over the estuary of the River Tay in Scotland collapsed in 1879, killing 75 people. A commission conducted a rigorous investigation that identified a number of contributing factors, including failure of the bridge designer to allow for wind loading and shoddy quality control over the manufacturing of the ironwork. The commission's recommendations, based on mathematical models and empirical studies of wind speeds and pressures, were immediately adopted by engineers designing a new rail bridge over the Forth estuary near Edinburgh. That bridge, the longest cantilever bridge in the world when completed in 1890 and still the second longest, has stood for over 120 years and carries about two hundred trains daily. The lesson is clear: while mathematics alone will not guarantee your bridge will stand, mathematics, if properly applied, can guarantee the bridge will stand if its construction, environment, and use are as modeled in the design.

That mathematics can guide and reassure the engineer is nothing new (though the fact that symbolic mathematics can model the physical world is really awe-inspiring if one just stops taking it for granted). Engineers have been using mathematics this

way for millennia. But as our mathematical and computational tools grow more sophisticated, they find more use in engineering.

I have witnessed this firsthand in hardware and software verification. My own specialty is the construction and use of mechanical theorem provers, i.e., software that attempts to prove a formula by deriving it from a small set of axioms using a small set of inference rules like chaining together previously proved results, substitution of equals for equals, and induction. If the axioms are all valid ("always true") and if the rules of inference preserve validity, then any formula so proved must be valid. Thus, proof is a way to establish truth. Mathematicians have been using the axiomatic method since Euclid.

A *theorem* is a formula that has been proved. A *lemma* is just another name for a theorem, but the label *lemma* is generally used only for formulas whose primary role is in the proofs of more interesting formulas. If Lemma 1 tells us that when P is true then Q is true, and Lemma 2 tells us that when Q is true, R is true, then we can chain them together to prove the theorem that when P is true, R is true.

But how do we know that a "proof" is a proof? Traditionally, when a mathematician publicizes a proof, other mathematicians interested in that result scrutinize it, frequently finding flaws, typos, forgotten cases, unstated assumptions, etc. Over time these flaws are fixed—if possible—and eventually the mathematical community accepts the formula as proved.

Mechanical theorem provers are designed to circumvent this "social process" to some extent. If a result has been proved by a trusted mechanical prover, then one can rest assured that the "proof" is a proof and that the formula is "always true." The social process need only inspect the formula itself and its underlying definitions and decide whether the formula actually means what the author intended to say.

An important application of mechanical theorem provers is to prove properties of computer hardware and software. Exhaustive testing is impractical for modern computing artifacts: there are just too many cases to consider. Proof is the obvious way to establish the truth of properties. But the mathematical social process is not well suited to these proofs: the proofs are often long; there are myriad cases to consider because the artifacts, their specifications, and the underlying logical concepts are often complicated; the various logical concepts are related in a wide variety of ways by many lemmas; and the artifacts and their properties may be proprietary. Just keeping track of all the assumptions and known relationships can be an almost impossible challenge. Using a machine to check proofs is an ideal solution.

I said above that I've witnessed firsthand the slow acceptance of "new" mathematical methods by the engineering community. Here's part of the story:

In 1987, after 16 years of working on mechanical theorem provers for hardware and software verification, my colleagues and I at the University of Texas at Austin started a company whose mission was to spread the technology to industry. One of those colleagues was David Russinoff, who joined our theorem proving group in 1983. When we learned that David had a PhD in number theory from the Courant Institute, we challenged him to prove Wilson's Theorem with the prover and offered him a master's degree for doing so. He succeeded and went on to

prove mechanically many other results in number theory, including Gauss's Law of Quadratic Reciprocity. His was the first mechanically checked proof of that theorem and, as he writes in his online bibliography, "The primary significance of this last result, of course, was that it finally put to rest any suspicion that the 197 previously published proofs of this theorem were all flawed." His humor obscures the fact that the result was of sufficient interest that 197 proofs were worthy of publication!

The company that the UT Austin group spun off in 1987 was called Computational Logic, Inc (CLI). There we continued to develop our tools, and in 1989, together with my colleagues Bob Boyer and Matt Kaufmann, I started work on a new prover that we called ACL2: A Computational Logic for Applicative Common Lisp. The characteristics of ACL2 are unimportant except for three things. First, ACL2's logic is a subset of a standard programming language and is thus well suited to modeling computational artifacts like hardware designs, programming languages, algorithms, etc. Second, the more complicated a proof is, the more help the ACL2 user has to provide. Think of the prover as trying to find a path from point A to point B. The more intermediate milestones the user provides, the more successful the prover will be. Those milestones are the key lemmas in the proof. Third, the existence of prior ACL2 work in a domain can be a great aid to the user because it may not be necessary to formalize and prove so many lemmas.

In 1995, Advanced Micro Devices (AMD) hired the company to prove the mathematical correctness of the microcode for its floating-point division operation on the soon-to-be-fabricated AMD K5 microprocessor. This was especially important because the Intel Pentium FDIV bug was just beginning to be publicized and AMD had recently made major changes in its floating-point unit requiring the design team to discard previously tested hardware for division and re-implement it in microcode using floating-point addition, subtraction, and multiplication.

I temporarily joined the AMD floating-point design team to lead the CLI proof effort. The goal was to prove that the FDIV algorithm complied with IEEE Floating-Point Standard 754. Roughly speaking, the tasks could be thought of as follows: formalize the relevant part of the standard, formalize the algorithm, formalize the desired properties, and then lead the mechanical theorem prover to a proof of the properties by discovering, formalizing, and piecing together lemmas about floating-point operations and concepts such as rounding. ACL2 had never been used to do proofs about floating point so there was no "bookshelf" of previously proved lemmas to draw upon. And while I had been programming for decades and thus had a passing familiarity with floating point, I was basically ignorant of the technical details. The designers in the group, many with a decade or more of experience in floating-point design, were the experts. But their explanations of the various steps taken by the division algorithm often involved examples, graphs plotting trends in error reduction, assertions that this or that technique was well known and had been reliably used in earlier products, etc. My real job was to distill that informal "shop talk" down into formulas and prove them.

Once in a weekly meeting, when asked what I'd accomplished that week, I displayed a lemma I'd proved only to be met by the universal response "we knew that." However, there is a big difference between "knowing" something

and writing it down precisely so that it is always true even when interpreted by the most malicious of readers! The lemma under discussion was a version of Lemma 6.85 concerning what was called “sticky” rounding. I strongly suspect that while everybody in the room could state some of the necessary hypotheses, I was the only person in the room who could state them all and do so with confidence.

Working with Matt Kaufmann and the lead designer, Tom Lynch, we eventually got ACL2 to prove the theorem in question. The project took a total of 9 weeks and the K5 was fabricated on time.

AMD was interested in proving other properties of components of its floating-point unit, but it was uninterested in hiring CLI to do so: sharing AMD proprietary designs with outsiders was unusual, to say the least. Aside from offering AMD assurance that its FDIV operation was correct, AMD considered the CLI project a test to see if our technology was useful. We passed the test, but they wanted an AMD employee to drive the prover. The person they hired away from CLI was David Russinoff.

One might have hoped that upon arrival at AMD, David could simply build on the library of lemmas developed for the FDIV proof. But that library was just what you’d expect from a 9-week crash effort: an ad hoc collection of inelegant formulas. Each was, of course, valid—that is the beauty of mechanized proof—but they were far from a useful theory of floating point. So David started over and followed a disciplined approach to developing such a theory in ACL2.

Here, he presents a carefully considered collection of the key properties of floating-point operations of use when proving a wide variety of theorems about many different kinds of algorithms. The properties are stated accurately and in complete detail, there are no hidden assumptions, and each lemma and theorem is valid. I consider this work a *tour de force* in formal reasoning about floating-point designs. It truly provides a formal mathematical basis for the analysis of floating point.

The definitions and theorems in this book are shown in conventional mathematical notation, not the rigid formal syntax of ACL2. But underlying this volume is a large collection of definitions and theorems in ACL2’s syntax, all formally processed and certified by ACL2. That collection—built by David over 20 years of industrial application of ACL2 to floating-point designs—is a powerful aid to formal verification. Many familiar companies have used David’s library. This book is exquisite documentation of that library.

If you trust ACL2 and care only about validity, the proofs shown here are unimportant, since the formulas have been mechanically verified. But in a much deeper sense, following the traditions of mathematics, the proofs are everything because *they explain why these formulas are valid*. Each proof also illustrates how the previous lemmas can be used—a point driven home mainly by Part V of the book, where David formally analyzes a commercially interesting floating-point unit.

Returning to the general theme of this foreword—the role of mathematics in the construction of useful artifacts and the adoption of techniques that were impractical just decades earlier—it is not unusual today to see logic-based tools, such as mechanized theorem provers, in use in design and verification groups in

the microprocessor industry. Companies such as AMD, Arm, Centaur, Intel, Oracle, Samsung, and Rockwell Collins have used mechanical theorem provers to prove important theorems about their products.

That last sentence would have been unthinkable just decades ago. Formal logic, once studied only by philosophers and logicians, now is a branch of applied mathematics. Mechanized provers are used in the design of amazing but now common everyday objects found in everything from high-end servers to mobile devices and from medical instruments to aircraft avionics. Indeed, many modern microprocessor designs simply cannot be built with confidence without such tools. The work presented in this book makes it easier to do the once unimaginable.

Edinburgh, Scotland J Strother Moore
April 2018

Preface

It is not the purpose of this book to expound the principles of computer arithmetic algorithms, nor does it presume to offer instruction in the art of arithmetic circuit design. A variety of publications spanning these subjects are readily available, including general texts as well as more specialized treatments, covering all areas of functionality and aspects of implementation. There is one relevant issue, however, that remains to be adequately addressed: the problem of eliminating human error from arithmetic hardware designs and establishing their ultimate correctness.

As in all areas of computer architecture, the designer of arithmetic circuitry is preoccupied with efficiency. His objective is the rapid development of logic that optimizes resource utilization and maximizes execution speed, guided by established practices and intuition. Subtle conceptual errors and miscalculations are accepted as inevitable, with the expectation that they will be eliminated through a separate validation effort.

As implementations grow in complexity through the use of increasingly sophisticated techniques, errors become more difficult to detect. It is generally acknowledged that testing alone is insufficient to provide a satisfactory level of confidence in the functional correctness of a state-of-the-art floating-point unit; formal verification methods are now in widespread use. A common practice is the use of an automated sequential logic equivalence checker [20, 34] to compare a proposed register-transfer logic (RTL) design either to an older trusted design or to a high-level C++ model. One deficiency of this approach is that the so-called golden model, whether coded in Verilog or C++, has typically never been formally verified itself and thus cannot be guaranteed to be free of errors. Another is the inherent complexity limitations of such tools [33], which have been found to render them inadequate for the comprehensive verification of complex high-precision floating-point modules.

A variety of projects have attempted to address these issues by combining such automatic methods with the power of interactive mechanical theorem proving [21, 25]. This book is an outgrowth of one such effort in the formal verification of commercial floating-point units, conducted over the course of two decades during which I was employed by Advanced Micro Devices, Inc. (1996–2011), Intel Corp. (2012–2016), and Arm Holdings (2016–present). My theorem prover of choice

is ACL2 [11–13], a heuristic prover based on first-order logic, list processing, rational arithmetic, recursive functions, and mathematical induction. ACL2 is a freely available software system, developed and maintained at the University of Texas by Matt Kaufmann and J Moore.

The principal advantages of theorem proving over equivalence and model checking are greater flexibility, derived from a more expressive underlying logical notation, and scalability to more complex designs. The main drawback is the requirement of more control and expertise on the part of the user. In the domain of computer arithmetic, effective use of an interactive prover entails a thorough understanding and a detailed mathematical exposition of the design of interest. Thus, the success of our project requires an uncommonly scrupulous approach to the design and analysis of arithmetic circuits. Loose concepts, intuition, and arguments by example must be replaced by formal development, explicit theorems, and rigorous proofs.

One problem to be addressed is the semantic gap between abstract behavioral specifications and concrete hardware models. While the design of a circuit is modeled for most purposes at the bit level, its prescribed behavior is naturally expressed in terms of high-level arithmetic concepts and algorithms. It is often easier to prove the correctness of an algorithm than to demonstrate that it has been implemented accurately.

As a simple illustration, consider the addition of two numbers, x and y, as diagrammed below in binary notation. Suppose the sum $z = x + y$ is to be truncated at the dotted line and that the precision of x is such that its least significant nonzero bit lies to the left of the line.

```
   1xxxxx.xxxxxxxxx|00 ···
+    1yyy.yyyyyyyyy|yy ···
---------------------------
  1zzzzzz.zzzzzzzzz|zz ···
```

Instead of computing the exact value of z and then extracting the truncated result, an implementation may choose to perform the truncation on y instead (at the same dotted line) before adding it to x. The designer, in order to convince himself of the equivalence of these two approaches, might resort to a diagram like the one above.

The next logical step would be to formulate the underlying principle in precise terms (see Lemma 6.14 in Sect. 6.1), explicitly identifying the necessary conditions for equivalence, and establishing its correctness by rigorous mathematical proof. The result could then be integrated into an evolving theory of computer arithmetic and thus become available for reuse and subject to extension and generalization (e.g., Lemma 6.97 in Sect. 6.5) as appropriate for new applications.

Such results are nowhere to be found in the existing literature, which is more concerned with advanced techniques and optimizations than with their theoretical underpinnings. What prevents the organization of the essential properties of the basic data objects and operations of this domain into a theory suitable for systematic

application? The foregoing example, however trivial, serves to illustrate one of the main obstacles to this objective: the modeling of data at different levels of abstraction. Numbers are naturally represented as bit vectors, which, for some purposes, may be viewed simply as strings of Boolean values. Truncation, for example, is conveniently described as the extraction of an initial segment of bits. For the purpose of analyzing arithmetic operations, on the other hand, the same data must be interpreted as abstract numbers. Although the correspondence between a number and its binary representation is straightforward, a rigorous proof derived from this correspondence requires more effort than an appeal to intuition based on a simple diagram. Consequently, the essential properties of bit vector arithmetic have never been formalized and compiled into a well-founded comprehensive theory. In the absence of such a theory, the designer must rediscover the basics as needed, relying on examples and intuition rather than theorems and proofs.

Nearly two centuries ago, the Norwegian mathematician Niels Abel complained about a similar state of affairs in another area of mathematical endeavor [14]:

> It lacks so completely all plan and system that it is peculiar that so many men could have studied it. The worst is, it has never been treated stringently. There are very few theorems ... which have been demonstrated in a logically tenable manner. Everywhere one finds this miserable way of concluding from the specific to the general ...

The subject in question was the calculus, a major mathematical development with a profound impact on the sciences, but lacking a solid logical foundation. Various attempts to base it on geometry or on intuition derived from other areas of mathematics proved inadequate as the discipline grew in complexity. The result was a climate of uncertainty, controversy, and stagnation. Abel and others resolved to restore order by rebuilding the theory of calculus solely on the basis of arithmetical concepts, thereby laying the groundwork for modern mathematical analysis.

While the contemporary hardware engineer may not be susceptible to the same philosophical qualms that motivated nineteenth-century mathematicians, he is certainly concerned with the "bugs" that inevitably attend undisciplined reasoning. Even if the analogy overreaches the present problem, it charts a course for its solution and sets the direction of this investigation.

Contents and Structure of the Book

Our initial objective is a unified mathematical theory, derived from the first principles of arithmetic, encompassing two distinct domains of discourse. The first of these, which is the subject of Part I, is the realm of register-transfer logic (RTL), comprising the primitive data types and operations on which microprocessor designs are built: bit vectors and logical operations. A critical first step is the careful formulation of these primitives in a manner consistent with our goals, which sometimes requires resistance to intuition. Thus, notwithstanding its name, we define a bit vector of width k to be a natural number less than 2^k rather than a sequence of k Boolean values. This decision will seem unnatural to those who

are accustomed to dealing with these objects in the context of hardware description languages, but it is a critical step in the master plan of an arithmetic-based theory. As a consequence, which may also be disturbing to some, the logical operations are defined recursively rather than bit-wise.

Part II addresses the second domain of interest, the more abstract world of rational arithmetic, focusing on floating-point representations of rational numbers as bit vectors. The benefits of a rigorous approach are most evident in the chapter on floating-point rounding, which includes a variety of results that would otherwise be difficult to state or prove, especially those that relate abstract arithmetic operations on rationals to lower-level properties of bit vectors. All of the architectural rounding modes prescribed by IEEE Standard 754 [9] are thoroughly analyzed. Moreover, since hardware implementation is of central interest, further attention is given to several other modes that are commonly used for internal computations but are not normally covered in treatises on floating-point arithmetic.

In Part III, the theory is extended to the analysis of several well-known algorithms and techniques used in the implementation of elementary arithmetic operations. The purpose here is not to present a comprehensive survey of the field, but merely to demonstrate a methodology for proving the correctness of implementations, providing guidance to those who are interested in applying it further. There is a chapter on addition, including a discussion of leading zero anticipation, and another on multiplication, describing several versions of Booth encoding. Two division algorithms are analyzed: a subtractive SRT algorithm, which is also applied to square root extraction, and a multiplicative algorithm based on a fused multiplication-addition operation.

Although IEEE 754 is routinely cited as a specification of correctness of floating-point implementations, it contains a number of ambiguities and leaves many aspects of behavior unspecified, as reflected in the divergent behaviors exhibited by various "compliant" architectures, especially in the treatment of exceptional conditions. In particular, the two primary floating-point instruction sets of the x86 architecture, known as *SSE* and *x87*, employ distinct exception-handling procedures that have been implicitly established across the microprocessor industry. Another important floating-point instruction set, with its own variations of exception handling, is provided by the Arm architecture. For every new implementation of any of these architectures, backward compatibility is a strict requirement. Unfortunately, no existing published reference is adequate for this purpose. When a verification engineer seeks clarification of an architectural detail, he is likely to consult an established expert, who may refer to a trusted RTL module or even a comment embedded in a microcode file, but rarely a published programming manual and never, in my experience, an IEEE standard. We address this problem in Part IV, presenting comprehensive behavioral specifications for the elementary arithmetic instructions of these three instruction sets—SSE, x87, and Arm—which were compiled and tested over the course of more than twenty years through simulation and analysis of commercial RTL models.

Part V describes and illustrates our verification methodology. In Chap. 15, we present a functional programming language, essentially a primitive subset of C

augmented by C++ class templates that implement integer and fixed-point registers. This language has proved suitable for abstract modeling of floating-point RTL designs and is susceptible to mechanical translation to the ACL2 logic. The objective in coding a design in the language is a model that is sufficiently faithful to the RTL to allow efficient equivalence checking by a standard commercial tool, but as abstract as possible in order to facilitate formal analysis. The ACL2 translation of such a model may be verified to comply with the appropriate behavioral specification of Part IV as an application of the results of Parts I–III, thereby establishing correctness of the original RTL. Each of Chaps. 16–19 contains a correctness proof of a module of a state-of-the-art floating-point unit that has been formally verified through this process.

Formalization: The Role of ACL2

All definitions, lemmas, and theorems presented in this exposition have been formalized in the logical language of ACL2 and mechanically checked with the ACL2 prover. The results of Parts I–IV have been collected in an evolving library—a component of the standard ACL2 release—which has been used in the formal verification of a variety of arithmetic RTL designs [23, 27–32]. These include the modules presented in Part V, the proof scripts for which also reside in the ACL2 repository, and are thus available to the ACL2 user, who may wish to "replay" and experiment with these proofs on his own machine.

The role of ACL2 in the development of the theory is evidenced in various ways throughout the exposition, including its emphasis on recursion and induction, but mainly in the level of rigor that is inherent in the logic and enforced by the prover. Any vague arguments, miscalculations, or missing hypotheses may be assumed to have been corrected through the mechanization process. With regard to style of presentation, the result will appeal to those readers whose thought processes most closely resemble the workings of a mechanical theorem prover; others may find it pedantic.

The above claim regarding the correspondence between the results presented here and their ACL2 formalizations requires a caveat. Since the ACL2 logic is limited to rational arithmetic,[1] properties that hold generally for real numbers can be stated formally only as properties of rationals. In the chapters relevant to the theory of rounding—Chaps. 1, 4, and 6—all results, with the single exception of Lemma 4.13, are stated and proved more generally, with real variables appearing in place of the rational variables found in the corresponding ACL2 code.

This limitation of the logic is especially relevant to the square root operator, which cannot be defined as an ACL2 function. This presents a challenge in the formalization of IEEE compliance (i.e., correctly rounded results) of a square root

[1]It should be noted that an extension of ACL2 supporting the reals through nonstandard analysis has been implemented by Gamboa [6].

implementation, which is addressed in Chap. 7. Here, we introduce an equivalent formulation of the IEEE requirement that is confined to the domain of rational arithmetic and thus provides for the ACL2 formalization of the architectural specifications presented in Part IV.

On the other hand, the ACL2 connection may be safely ignored by those uninterested in the formal aspect of the theory. Outside of Sect. 15.6, no familiarity with ACL2, LISP, or formal logic is required of (or even useful to) the reader. Several results depend on computations that have been executed with ACL2 and cannot reasonably be expected to be carried out by hand,[2] but in each case, the computation is explicitly specified and may be readily confirmed in any suitable programming language. Otherwise, the entire exposition is surveyable and self-contained, adhering to the most basic conventions of mathematical notation, supplemented only by several RTL constructs common to hardware description languages, all of which are explicitly defined upon first use. Nor is any uncommon knowledge of mathematics presupposed. The entire content should be accessible to a competent high school student who has been exposed to the algebra of real numbers and the principle of mathematical induction, especially one with an assiduous capacity for detail. It must be conceded, however, that repeated attempts to substantiate this claim have been consistently unsuccessful.

The book has been written with several purposes in mind. For the ACL2 user interested in applying or extending the associated RTL library, it may be read as a user's manual. The theory might also be used to guide other verification efforts, either without mechanical support or encoded in the formalism of another theorem prover. A more ambitious goal is a rigorous approach to arithmetic circuit design that is accessible and useful to architects and RTL writers.

Obtaining the Associated ACL2 Code

The ACL2 distribution [13] includes a `books` directory, consisting of libraries contributed and maintained by members of the ACL2 user community. Three of its subdirectories are related to this project:

- `books/rtl/`: The formalization of the theory presented in Parts I–IV;
- `books/projects/rac/`: The parser and ACL2 translator for the language described in Chap. 15;
- `books/projects/arm`: The scripts for the proofs presented in Chaps. 16–19.

A more up-to-date version of the `books` directory is available more directly through the GitHub hosting service at

`https://github.com/acl2/acl2/tree/master/books/.`

[2]These pertain to table-based reciprocal computation (Lemma 11.1), FMA-based division (Lemmas 11.7 and 11.9), and SRT division and square root (Lemmas 10.7, 10.8, and 10.15).

Acknowledgments

In 1995, in the wake of the Intel FDIV affair, Moore and Kaufmann were contracted by AMD to verify the correctness of the division instruction of the K5 processor—AMD's answer to the Pentium—with their new theorem prover. Soon thereafter, I was assigned the task of extending their work to the K5 square root operation. I am indebted to them for creating a delightful prover and demonstrating its effectiveness as a floating-point verification tool, thus laying a foundation for this venture.

Whatever I learned about arithmetic circuits over the intervening twenty-two years was explained to me by the designers with whom I have been privileged to collaborate: Mike Achenbach, Javier Bruguera, Michael Dibrino, David Dean, Steve Espy, Warren Ferguson, Kelvin Goveas, Carl Lemonds, Dave Lutz, Tom Lynch, Stuart Oberman, Simon Rubanovich, and Peter Seidel.

A critical component of the AMD effort was a Verilog-ACL2 translator originally implemented by Art Flatau and me and later refined by Kaufmann, Hanbing Liu, and Rob Sumners. The ACL2 formalization of some of the early proofs was done by Kaufmann, Liu, and Eric Smith.

The development and application of the verification methodology described in Part V began at Intel and has continued at Arm. The initial design of the modeling language was a collaboration with John O'Leary and benefited from suggestions by Rubanovich. Bruguera and Lutz are the architects of the Arm FPU on which Chaps. 16–19 are based.

I am also grateful to Matthew Bottkol, Warren Ferguson, Cooky Goldblatt, David Hardin, Linda Ness, and Coke Smith for their suggestions for improvement of earlier versions of the manuscript and to Glenn Downing for telling me that it was time to publish the book. Finally, I cannot deny myself the pleasure of thanking Lin Russinoff, who designed the cover graphics and is my continual source of inspiration.

Austin, TX, USA David M. Russinoff
April 2018

Contents

Part I Register-Transfer Logic

1 Basic Arithmetic Functions 3
1.1 Floor and Ceiling 3
1.2 Modulus 6
1.3 Truncation 12

2 Bit Vectors 17
2.1 Bit Slices 18
2.2 Bit Extraction 24
2.3 Concatenation 28
2.4 Integer Formats 32
2.5 Fixed-Point Formats 35

3 Logical Operations 39
3.1 Binary Operations 39
3.2 Complement 46
3.3 Algebraic Properties 49

Part II Floating-Point Arithmetic

4 Floating-Point Numbers 53
4.1 Decomposition 53
4.2 Exactness 56

5 Floating-Point Formats 63
5.1 Classification of Formats 63
5.2 Normal Encodings 65
5.3 Denormals and Zeroes 69
5.4 Infinities and NaNs 74

6 Rounding 77
6.1 Rounding Toward Zero 79
6.2 Rounding Away from Zero 86

6.3 Rounding to Nearest 93
6.4 Odd Rounding 109
6.5 IEEE Rounding 117
6.6 Denormal Rounding 127

7 IEEE-Compliant Square Root 135
7.1 Truncated Square Root 136
7.2 Odd-Rounded Square Root 138
7.3 IEEE-Rounded Square Root 141

Part III Implementation of Elementary Operations

8 Addition 147
8.1 Bit Vector Addition 147
8.2 Leading Zero Anticipation 155
8.3 Trailing Zero Anticipation 160

9 Multiplication 165
9.1 Radix-2 Booth Encoding 166
9.2 Radix-4 Booth Encoding 167
9.3 Encoding Carry-Save Sums 173
9.4 Statically Encoded Multiplier Arrays 176
9.5 Radix-8 Booth Encoding 179

10 SRT Division and Square Root 183
10.1 SRT Division 183
10.2 Minimally Redundant Radix-4 Division 188
10.3 Minimally Redundant Radix-8 Division 190
10.4 SRT Square Root 191
10.5 Minimally Redundant Radix-4 Square Root 198

11 FMA-Based Division 203
11.1 Reciprocal Approximation 205
11.2 Quotient Refinement 207
11.3 Reciprocal Refinement 213
11.4 Examples 214

Part IV Comparative Architectures: SSE, x87, and Arm

12 SSE Floating-Point Instructions 221
12.1 SSE Control and Status Register 221
12.2 Overview of SSE Floating-Point Exceptions 222
12.3 Pre-computation Exceptions 223
12.4 Computation 224
12.5 Post-Computation Exceptions 225

13 x87 Instructions ... 227
- 13.1 x87 Control Word ... 227
- 13.2 x87 Status Word ... 228
- 13.3 Overview of x87 Exceptions ... 229
- 13.4 Pre-computation Exceptions ... 230
- 13.5 Post-Computation Exceptions ... 231

14 Arm Floating-Point Instructions ... 233
- 14.1 Floating-Point Status and Control Register ... 234
- 14.2 Pre-computation Exceptions ... 235
- 14.3 Post-Computation Exceptions ... 236

Part V Formal Verification of RTL Designs

15 The Modeling Language ... 239
- 15.1 Language Overview ... 240
- 15.2 Parameter Passing ... 242
- 15.3 Registers ... 243
- 15.4 Arithmetic ... 245
- 15.5 Control Restrictions ... 246
- 15.6 Translation to ACL2 ... 248

16 Double-Precision Multiplication ... 253
- 16.1 Parameters ... 254
- 16.2 Booth Multiplier ... 256
- 16.3 Unrounded Product ... 257
- 16.4 FMA Support ... 267
- 16.5 Rounded Product and FMUL ... 269

17 Double-Precision Addition and FMA ... 279
- 17.1 Parameters and Input Assumptions ... 279
- 17.2 Alignment ... 285
- 17.3 Addition ... 287
- 17.4 Leading Zero Anticipation ... 292
- 17.5 Normalization ... 294
- 17.6 Rounding ... 296
- 17.7 Correctness Theorems ... 302

18 Multi-Precision Radix-4 SRT Division ... 309
- 18.1 Overview ... 309
- 18.2 Pre-processing ... 311
- 18.3 Iterative Phase ... 314
- 18.4 Post-Processing and Rounding ... 320

19 Multi-Precision Radix-4 SRT Square Root ... 331
- 19.1 Pre-processing ... 331
- 19.2 Iterative Phase ... 333
- 19.3 Post-Processing and Rounding ... 337

Appendices 345

Bibliography 379

Index 381

Part I
Register-Transfer Logic

An arithmetic circuit design may be modeled at various levels of abstraction, from a *netlist*, which specifies an interconnection of electronic components, to a high-level numerical algorithm. Although the former is required for implementation, modern synthesis tools can readily create such a model from a higher-level *register-transfer logic* (RTL) design coded in a hardware description language (HDL) such as Verilog. There also exist high-level synthesis tools that can convert numerical algorithms coded in C++ to Verilog RTL, but these are not nearly mature enough to achieve the level of efficiency required of today's commercial floating-point units. Consequently, most of the design and verification effort is conducted at the RTL level, i.e., in terms of the flow of data between hardware registers and the logical operations performed on them.

The following three chapters may be viewed as an exposition of basic HDL semantics. The fundamental data type is the *bit vector*, which is implemented as an ordered set of two-state devices. A possible starting point for our theory, therefore, is the definition of a bit vector as a sequence of Boolean values. An advantage of this approach is that it allows straightforward definitions of primitive operations such as bit extraction and concatenation. In the realm of computer arithmetic, however, a bit vector is viewed more fruitfully as the binary expansion of an integer. In our formalization, we shall identify a bit vector with the number that it so represents, i.e., we shall define a bit vector to be an integer. This naturally leads to the definition of the primitive RTL operations as arithmetic functions. While the consequences of this decision may sometimes seem cumbersome, its benefits will become clear in later chapters as we further explore the arithmetic of bit vectors.

In Chap. 2, we formalize and examine the properties of bit vectors and the primitive operations of bit slice, bit extraction, and concatenation. The bit-wise logical operations are discussed in Chap. 3. All of these operations are defined in terms of the basic arithmetic functions *floor* and *modulus*, which are the subject of Chap. 1.

Chapter 1
Basic Arithmetic Functions

This chapter examines the properties of the *floor*, *ceiling*, and *modulus* functions, which are central to our formulation of the RTL primitives as well as the floating-point rounding modes. Thus, their definitions and properties are prerequisite to a reading of the subsequent chapters. We also define and investigate the properties of a function that truncates to a specified number of fractional bits, which is related to the floor and is relevant to the analysis of fixed-point encodings, as discussed in Sect. 2.5.

The reader will find most of the lemmas of this chapter to be self-evident and may question the need to include them all, but it will prove convenient to have these results collected for later reference.

Notation The symbols $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$, and $\mathbb{Z}^+$ will denote the sets of all real numbers, rational numbers, integers, natural numbers (i.e., nonnegative integers), and positive integers, respectively.

1.1 Floor and Ceiling

The functions $\lfloor x \rfloor$ and $\lceil x \rceil$, known as the *floor* and *ceiling*, are approximations of reals by integers. The floor is also known as the *greatest integer* function, because the value of $\lfloor x \rfloor$ may be characterized as the greatest integer not exceeding x:

Definition 1.1 For each $x \in \mathbb{R}$, $\lfloor x \rfloor$ is the unique integer that satisfies

$$\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1.$$

We list several obvious consequences of the definition:

© Springer Nature Switzerland AG 2019
D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_1

Lemma 1.1 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}$.*

(a) $\lfloor x \rfloor = x \Leftrightarrow x \in \mathbb{Z}$;
(b) $x \leq y \Rightarrow \lfloor x \rfloor \leq \lfloor y \rfloor$;
(c) $n \leq x \Rightarrow n \leq \lfloor x \rfloor$;
(d) $\lfloor x + n \rfloor = \lfloor x \rfloor + n$.

The following simplifying rule for quotients may be less obvious:

Lemma 1.2 *If $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$, then $\lfloor \lfloor x \rfloor / n \rfloor = \lfloor x/n \rfloor$.*

Proof Since $\lfloor x \rfloor \leq x$, $\lfloor x \rfloor / n \leq x/n$, and by monotonicity, $\lfloor \lfloor x \rfloor / n \rfloor \leq \lfloor x/n \rfloor$. To derive the reverse inequality, note that since $x/n \geq \lfloor x/n \rfloor$, we have $x \geq n \lfloor x/n \rfloor$. It follows that $\lfloor x \rfloor \geq n \lfloor x/n \rfloor$, and hence $\lfloor x \rfloor / n \geq \lfloor x/n \rfloor$, which implies $\lfloor \lfloor x \rfloor / n \rfloor \geq \lfloor x/n \rfloor$. □

The next result is used in a variety of inductive proofs pertaining to bit vectors. (See, for example, the proof of Lemma 2.40.)

Lemma 1.3 *If $n \in \mathbb{Z}$, then $|\lfloor n/2 \rfloor| \leq |n|$, and if $n \notin \{0, -1\}$, then $|\lfloor n/2 \rfloor| < |n|$.*

Proof It is clear that equality holds for $n = 0$ or $n = -1$. If $n \geq 1$, then

$$|\lfloor n/2 \rfloor| = \lfloor n/2 \rfloor \leq n/2 < n = |n|.$$

If $n \leq -2$, then

$$n/2 - 1 < \lfloor n/2 \rfloor \leq n/2 \leq -1$$

and therefore

$$|\lfloor n/2 \rfloor| = -\lfloor n/2 \rfloor < -(n/2 - 1) = -n/2 + 1 \leq -n = |n|.$$

□

The floor commutes with negation only for integer arguments:

Lemma 1.4 *For all $x \in \mathbb{R}$,*

$$\lfloor -x \rfloor = \begin{cases} -\lfloor x \rfloor & \text{if } x \in \mathbb{Z} \\ -\lfloor x \rfloor - 1 & \text{if } x \notin \mathbb{Z}. \end{cases}$$

Proof If $x \in \mathbb{Z}$, then Lemma 1.1 implies

$$\lfloor -x \rfloor = -x = -\lfloor x \rfloor.$$

Otherwise,

$$\lfloor x \rfloor < x < \lfloor x \rfloor + 1,$$

which implies

$$-\lfloor x \rfloor - 1 < -x < -\lfloor x \rfloor,$$

and by Definition 1.1,

$$\lfloor -x \rfloor = -\lfloor x \rfloor - 1.$$

□

When x is expressed as a ratio of integers, we also have the following unconditional expression for $\lfloor -x \rfloor$.

Lemma 1.5 *If $m \in \mathbb{Z}$, $n \in \mathbb{Z}$, and $n > 0$, then*

$$\lfloor -m/n \rfloor = -\lfloor (m-1)/n \rfloor - 1.$$

Proof Suppose first that $m/n \in \mathbb{Z}$. Then

$$\lfloor (m-1)/n \rfloor = \lfloor m/n - 1/n \rfloor = m/n + \lfloor -1/n \rfloor = m/n - 1$$

and

$$-\lfloor (m-1)/n \rfloor - 1 = -m/n = \lfloor -m/n \rfloor.$$

Now suppose $m/n \notin \mathbb{Z}$. Then $m/n > \lfloor m/n \rfloor$, which implies $m > \lfloor m/n \rfloor n$, and hence $m \geq \lfloor m/n \rfloor n + 1$. Thus,

$$\lfloor m/n \rfloor \leq (m-1)/n < m/n < \lfloor m/n \rfloor + 1,$$

and by Definition 1.1, $\lfloor (m-1)/n \rfloor = \lfloor m/n \rfloor$. Finally, by Lemma 1.4,

$$\lfloor -m/n \rfloor = -\lfloor m/n \rfloor - 1 = -\lfloor (m-1)/n \rfloor - 1.$$

□

Examples

$\lfloor -\frac{6}{5} \rfloor = -\lfloor \frac{6-1}{5} \rfloor - 1 = -\lfloor 1 \rfloor - 1 = -1 - 1 = -2$

$\lfloor -1 \rfloor = \lfloor -\frac{5}{5} \rfloor = -\lfloor \frac{5-1}{5} \rfloor - 1 = -\lfloor \frac{4}{5} \rfloor - 1 = 0 - 1 = -1$

$\lfloor -\frac{4}{5} \rfloor = -\lfloor \frac{4-1}{5} \rfloor - 1 = -\lfloor \frac{3}{5} \rfloor - 1 = 0 - 1 = -1$

The ceiling is defined most conveniently using the floor:

Definition 1.2 For all $x \in \mathbb{R}$, $\lceil x \rceil = -\lfloor -x \rfloor$.

We have an alternative characterization of $\lceil x \rceil$, analogous to Definition 1.1, as the least integer not exceeded by x:

Lemma 1.6 *For all* $x \in \mathbb{R}$, $\lceil x \rceil \in \mathbb{Z}$ *and* $\lceil x \rceil \geq x > \lceil x \rceil - 1$.

Proof By Definition 1.1, $\lfloor -x \rfloor \leq -x < \lfloor -x \rfloor + 1$, which leads to $-\lfloor -x \rfloor \geq x < -\lfloor -x \rfloor - 1$. The lemma now follows from Definition 1.2. □

We also have analogs of Lemmas 1.1 and 1.2:

Lemma 1.7 *Let* $x \in \mathbb{R}$, $y \in \mathbb{R}$, *and* $n \in \mathbb{Z}$.

(a) $\lceil x \rceil = x \Leftrightarrow x \in \mathbb{Z}$;
(b) $x \leq y \Rightarrow \lceil x \rceil \leq \lceil y \rceil$;
(c) $n \geq x \Rightarrow n \geq \lceil x \rceil$;
(d) $\lceil x + n \rceil = \lceil x \rceil + n$.

Lemma 1.8 *If* $x \in \mathbb{R}$ *and* $n \in \mathbb{Z}^+$, *then* $\lceil \lceil x \rceil / n \rceil = \lceil x/n \rceil$.

Proof By Definition 1.2 and Lemma 1.2,

$$\lceil \lceil x \rceil / n \rceil = -\lfloor -\lceil x \rceil / n \rfloor = -\lfloor \lfloor -x \rfloor / n \rfloor = -\lfloor -x/n \rfloor = \lceil x/n \rceil.$$

□

The floor and the ceiling are related as follows.

Lemma 1.9 *For all* $x \in \mathbb{R}$,

$$\lceil x \rceil = \begin{cases} \lfloor x \rfloor & \text{if } x \in \mathbb{Z} \\ \lfloor x \rfloor + 1 & \text{if } x \notin \mathbb{Z}. \end{cases}$$

Proof If $x \in \mathbb{Z}$, then of course, $\lceil x \rceil = \lfloor x \rfloor = x$. Otherwise, by Lemma 1.1, $x \neq \lfloor x \rfloor$, and hence Definition 1.1 yields $\lfloor x \rfloor < x < \lfloor x \rfloor + 1$. Rearranging these inequalities, we have $\lfloor x \rfloor + 1 > x > (\lfloor x \rfloor + 1) - 1$. By Lemma 1.6, $\lceil x \rceil = \lfloor x \rfloor + 1$. □

1.2 Modulus

The integer quotient of x and y may be defined as $\lfloor x/y \rfloor$. This formulation leads to the following characterization of the modulus function:

Definition 1.3 For all $x \in \mathbb{R}$ and $y \in \mathbb{R}$,

$$x \bmod y = \begin{cases} x - \lfloor x/y \rfloor y & \text{if } y \neq 0 \\ x & \text{if } y = 0. \end{cases}$$

Notation For the purpose of resolving ambiguous expressions, the precedence of this operator is higher than that of addition and lower than that of multiplication.

Although $x \bmod y$ is of interest mainly when $x \in \mathbb{Z}$ and $y \in \mathbb{Z}^+$, the definition is less restrictive and arbitrary real arguments must be considered. We note the following closure properties, which follow from Definitions 1.3 and 1.1.

Lemma 1.10 *Let $m \in \mathbb{Z}$ and $n \in \mathbb{Z}$.*

(a) $m \bmod n \in \mathbb{Z}$;
(b) *If* $n > 0$, *then* $m \bmod n \in \mathbb{N}$.

We have the following upper bounds in the integer case:

Lemma 1.11 *Let $m \in \mathbb{Z}$ and $n \in \mathbb{Z}$.*

(a) *If* $n > 0$, *then* $m \bmod n < n$;
(b) *If* $m \geq 0$, *then* $m \bmod n \leq m$;
(c) *If* $n > m \geq 0$, *then* $m \bmod n = m$.

Proof

(a) By Definitions 1.3 and 1.1,

$$m \bmod n = m - \lfloor m/n \rfloor n < m - ((m/n) - 1)n = n.$$

(b) By Definition 1.3, $m \bmod n = m - \lfloor m/n \rfloor n$. If $n > 0$, then $m/n > 0$, $\lfloor m/n \rfloor \geq 0$ by Lemma 1.1, and $\lfloor m/n \rfloor n \geq 0$. If $n < 0$, then $\lfloor m/n \rfloor \leq m/n \leq 0$, and again, $\lfloor m/n \rfloor n \geq 0$.
(c) Since $0 \leq m/n < 1$, $\lfloor m/n \rfloor = 0$ by Definition 1.1. Now by Definition 1.3,

$$m = \lfloor m/n \rfloor n + m \bmod n = m \bmod n.$$

□

Lemma 1.12 *If $a \in \mathbb{R}$ and $n \in \mathbb{R} - \{0\}$, then*

$$m \bmod n = 0 \Leftrightarrow m/n \in \mathbb{Z}.$$

Proof By Definition 1.3 and Lemma 1.1,

$$m \bmod n = 0 \Leftrightarrow m = \lfloor m/n \rfloor n \Leftrightarrow m/n = \lfloor m/n \rfloor \Leftrightarrow m/n \in \mathbb{Z}.$$

□

Lemma 1.13 *If $a \in \mathbb{Z}$, $b \in \mathbb{Z}$, and $n \in \mathbb{Z}^+$, then*

$$a \bmod n = b \bmod n \Leftrightarrow (a - b)/n \in \mathbb{Z}.$$

Proof By Definition 1.3,

$$a - b = \lfloor a/n \rfloor n - \lfloor b/n \rfloor n + (a \bmod n) - (b \bmod n).$$

Therefore,

$$(a-b)/n = \lfloor a/n \rfloor - \lfloor b/n \rfloor + ((a \bmod n) - (b \bmod n))/n$$

and

$$(a-b)/n \in \mathbb{Z} \Leftrightarrow ((a \bmod n) - (b \bmod n))/n \in \mathbb{Z}.$$

By Lemmas 1.10 and 1.11, $0 \le a \bmod n < n$ and $0 \le b \bmod n < n$, and hence,

$$((a \bmod n) - (b \bmod n))/n \in \mathbb{Z} \Leftrightarrow a = b.$$

□

Corollary 1.14 *Let $a \in \mathbb{Z}$, $b \in \mathbb{Z}$, and $n \in \mathbb{Z}$. If $|a-b| < n$, then*

$$a \bmod n = b \bmod n \Leftrightarrow a = b.$$

Proof Since $|(a-b)/n| < 1$, $(a-b)/n \in \mathbb{Z} \Leftrightarrow a = b$. The result follows from Lemma 1.13. □

Definition 1.4 For $a \in \mathbb{R}$, $b \in \mathbb{R}$, and $n \in \mathbb{R}$, a is congruent to b modulo n, or

$$a \equiv b \pmod{n},$$

if $a \bmod n = b \bmod n$.

According to Lemmas 1.12 and 1.13, if $a \in \mathbb{Z}$, $b \in \mathbb{Z}$, and $n \in \mathbb{Z}^+$, then

$$a \equiv b \pmod{n} \Leftrightarrow a - b \equiv 0 \pmod{n} \Leftrightarrow (a-b)/n \in \mathbb{Z}.$$

Lemma 1.15 *For all $a \in \mathbb{Z}$, $m \in \mathbb{Z}$, and $n \in \mathbb{Z}$,*

$$m + an \equiv m \pmod{n}.$$

Proof By Definition 1.3 and Lemma 1.1,

$$\begin{aligned}(m+an) \bmod n &= m + an - \lfloor (m+an)/n \rfloor \\ &= m + an - \lfloor m/n \rfloor - a \\ &= m - \lfloor m/n \rfloor \\ &= m \bmod n.\end{aligned}$$

□

Corollary 1.16 *Let $a \in \mathbb{Z}$, $m \in \mathbb{Z}$, $n \in \mathbb{Z}$, and $r \in \mathbb{Z}$.*

(a) If $an \leq m < (a+1)n$, then $m \text{ mod } n = m - an$;
(b) If $an \leq m < an + r$, then $m \text{ mod } n < r$.

Proof By Lemmas 1.15 and 1.11, $an \leq m < (a+1)n$ implies

$$m \text{ mod } n = (m - an) \text{ mod } n = m - an,$$

and if $an \leq m < an + r$, then

$$m \text{ mod } n = (m - an) \text{ mod } n \leq m - an < r.$$

□

Lemma 1.17 *For all $m \in \mathbb{Z}$, $n \in \mathbb{Z}$, and $p \in \mathbb{Z}$,*

$$m \text{ mod } np = 0 \Leftrightarrow m \text{ mod } n = 0 \textit{ and } \lfloor m/n \rfloor \text{ mod } p = 0.$$

Proof By Lemma 1.13,

$$\begin{aligned} m \text{ mod } np = 0 &\Leftrightarrow m/(np) \in \mathbb{Z} \\ &\Leftrightarrow m/n \in \mathbb{Z} \text{ and } (m/n)/p \in \mathbb{Z} \\ &\Leftrightarrow m/n \in \mathbb{Z} \text{ and } \lfloor m/n \rfloor / p \in \mathbb{Z} \\ &\Leftrightarrow m \text{ mod } n = 0 \text{ and } \lfloor m/n \rfloor \text{ mod } p = 0. \end{aligned}$$

□

Lemma 1.18 *For all $k \in \mathbb{Z}$, $m \in \mathbb{Z}$, and $n \in \mathbb{Z}$,*

$$km \text{ mod } kn = k(m \text{ mod } n).$$

Proof By Definition 1.3,

$$\begin{aligned} km \text{ mod } kn &= km - \left\lfloor \frac{km}{kn} \right\rfloor kn \\ &= k(m - \lfloor m/n \rfloor n) \\ &= k(m \text{ mod } n). \end{aligned}$$

□

As another consequence of Lemmas 1.10 and 1.11, *mod* is an idempotent operator in the sense that for $n > 0$,

$$(m \text{ mod } n) \text{ mod } n = m \text{ mod } n.$$

This observation may be generalized as follows:

Lemma 1.19 *For all $m \in \mathbb{Z}$, $k \in \mathbb{Z}$, and $n \in \mathbb{Z}$,*

$$(m \bmod kn) \bmod n = m \bmod n.$$

Proof By Definition 1.3,

$$\begin{aligned}(m \bmod kn) \bmod n &= (x \bmod kn) - \lfloor (x \bmod kn)/n \rfloor n \\ &= x - \left\lfloor \frac{x}{kn} \right\rfloor kn - \left\lfloor \frac{x - \left\lfloor \frac{x}{kn} \right\rfloor kn}{n} \right\rfloor n \\ &= x - \left\lfloor \frac{x}{kn} \right\rfloor kn - \left\lfloor \frac{x}{n} - \left\lfloor \frac{x}{kn} \right\rfloor k \right\rfloor n \\ &= x - \left\lfloor \frac{x}{kn} \right\rfloor kn - \left(\left\lfloor \frac{x}{n} \right\rfloor - \left\lfloor \frac{x}{kn} \right\rfloor k \right) n \\ &= x - \left\lfloor \frac{x}{n} \right\rfloor n \\ &= x \bmod n.\end{aligned}$$

□

Lemma 1.19 is used most frequently with power-of-two moduli.

Corollary 1.20 *For all $a \in \mathbb{Z}$, $b \in \mathbb{Z}$, and $m \in \mathbb{Z}$, if $a \geq b \geq 0$, then*

$$(m \bmod 2^a) \bmod 2^b = m \bmod 2^b.$$

Proof This is the case of Lemma 1.19 with $n = 2^b$ and $k = 2^{a-b}$. □

Lemma 1.21 *Let $a \in \mathbb{Z}$, $m \in \mathbb{Z}^+$, and $n \in \mathbb{Z}^+$. Then*

$$\left\lfloor \frac{a \bmod mn}{n} \right\rfloor = \left\lfloor \frac{a}{n} \right\rfloor \bmod m.$$

Proof By Lemmas 1.3, 1.1, and 1.2,

$$\begin{aligned}\left\lfloor \frac{a \bmod mn}{n} \right\rfloor &= \left\lfloor \frac{a - mn \left\lfloor \frac{a}{mn} \right\rfloor}{n} \right\rfloor \\ &= \left\lfloor \frac{a}{n} \right\rfloor - m \left\lfloor \frac{a}{mn} \right\rfloor \\ &= \left\lfloor \frac{a}{n} \right\rfloor - m \left\lfloor \frac{\left\lfloor \frac{a}{n} \right\rfloor}{m} \right\rfloor \\ &= \left\lfloor \frac{a}{n} \right\rfloor \bmod m.\end{aligned}$$

□

Lemma 1.22 *For all $a \in \mathbb{Z}$, $b \in \mathbb{Z}$, and $n \in \mathbb{Z}$,*

$$(a + (b \bmod n)) \bmod n = (a + b) \bmod n.$$

Proof By Definition 1.3, $b \bmod n = b - \lfloor b/n \rfloor n$, and hence

$$\begin{aligned}(a + (b \bmod n)) \bmod n &= a + (b \bmod n) - \lfloor (a + (b \bmod n))/n \rfloor n \\ &= a + b - \lfloor b/n \rfloor n - \lfloor (a + b)/n - \lfloor b/n \rfloor \rfloor n \\ &= a + b - \lfloor b/n \rfloor n - (\lfloor (a + b)/n \rfloor - \lfloor b/n \rfloor)n \\ &= a + b - \lfloor (a + b)/n \rfloor n \\ &= (a + b) \bmod n.\end{aligned}$$

□

Lemma 1.23 *For all $a \in \mathbb{Z}$, $b \in \mathbb{Z}$, and $n \in \mathbb{Z}$,*

$$(a - (b \bmod n)) \bmod n = (a - b) \bmod n.$$

Proof By Definition 1.3, $b \bmod n = b - \lfloor b/n \rfloor n$, and hence

$$\begin{aligned}(a - (b \bmod n)) \bmod n &= a - (b \bmod n) - \lfloor (a - (b \bmod n))/n \rfloor n \\ &= a - b + \lfloor b/n \rfloor n - \lfloor (a - b)/n + \lfloor b/n \rfloor \rfloor n \\ &= a - b + \lfloor b/n \rfloor n - (\lfloor (a - b)/n \rfloor + \lfloor b/n \rfloor)n \\ &= a - b - \lfloor (a - b)/n \rfloor n \\ &= (a - b) \bmod n.\end{aligned}$$

□

Lemma 1.24 *For all $a \in \mathbb{N}$, $b \in \mathbb{N}$, and $n \in \mathbb{Z}^+$, then*

$$(a \bmod n)b \bmod n = ab \bmod n.$$

Proof By Definition 1.3 and Lemma 1.1,

$$\begin{aligned}(a \bmod n)b \bmod n &= (a \bmod n)b - \left\lfloor \frac{(a \bmod n)b}{n} \right\rfloor n \\ &= \left(a - \left\lfloor \frac{a}{n} \right\rfloor n\right) b - \left\lfloor \frac{\left(a - \left\lfloor \frac{a}{n} \right\rfloor n\right) b}{n} \right\rfloor n \\ &= ab - \left\lfloor \frac{a}{n} \right\rfloor nb - \left\lfloor \frac{ab}{n} - \left\lfloor \frac{a}{n} \right\rfloor b \right\rfloor n\end{aligned}$$

$$= ab - \left\lfloor \frac{a}{n} \right\rfloor nb - \left(\left\lfloor \frac{ab}{n} \right\rfloor - \left\lfloor \frac{a}{n} \right\rfloor b \right) n$$
$$= ab - \left\lfloor \frac{ab}{n} \right\rfloor n$$
$$= ab \bmod n.$$

□

Lemma 1.25 *For all $a \in \mathbb{N}$, $b \in \mathbb{N}$, $c \in \mathbb{N}$, and $n \in \mathbb{Z}^+$, if $a \equiv b \pmod{n}$, then*

$$a + c \equiv b + c \pmod{n}$$

and

$$ac \equiv bc \pmod{n}.$$

Proof By Lemma 1.22,

$$(a+c) \bmod n = ((a \bmod n)+c) \bmod n = ((b \bmod n)+c) \bmod n = (b+c) \bmod n,$$

and by Lemma 1.24,

$$ac \bmod n = (a \bmod n)c \bmod n = (b \bmod n)c \bmod n = bc \bmod n.$$

□

1.3 Truncation

The following function truncates a real number to a specified number of fractional bits:

Definition 1.5 For $x \in \mathbb{R}$ and $k \in \mathbb{Z}$,

$$x^{(k)} = \frac{\lfloor 2^k x \rfloor}{2^k}.$$

Note that according to Definition 1.3, an equivalent definition is

$$x^{(k)} = x - x \bmod 2^{-k}.$$

Example Let

$$x = \frac{51}{8} = 6 + \frac{3}{8}.$$

Then

$$x^{(2)} = \frac{\lfloor 2^2 x \rfloor}{2^2} = \frac{\left\lfloor \frac{51}{2} \right\rfloor}{4} = \frac{25}{4} = 6 + \frac{1}{4}.$$

Lemma 1.26 *If $x \in \mathbb{R}$, $m \in \mathbb{Z}$, $k \in \mathbb{Z}$, and $k \leq m$, then*

$$\left(x^{(k)}\right)^{(m)} = \left(x^{(m)}\right)^{(k)} = x^{(k)}.$$

Proof

$$\left(x^{(k)}\right)^{(m)} = \frac{\left\lfloor 2^m \cdot \frac{\lfloor 2^k x \rfloor}{2^k} \right\rfloor}{2^m} = \frac{\lfloor 2^{m-k} \lfloor 2^k x \rfloor \rfloor}{2^m} = \frac{2^{m-k} \lfloor 2^k x \rfloor}{2^m} = \frac{\lfloor 2^k x \rfloor}{2^k} = x^{(k)}$$

and by Lemma 1.2,

$$\left(x^{(m)}\right)^{(k)} = \frac{\left\lfloor 2^k \cdot \frac{\lfloor 2^m x \rfloor}{2^m} \right\rfloor}{2^k} = \frac{\left\lfloor \frac{\lfloor 2^m x \rfloor}{2^{m-k}} \right\rfloor}{2^k} = \frac{\left\lfloor \frac{2^m x}{2^{m-k}} \right\rfloor}{2^k} = \frac{\lfloor 2^k x \rfloor}{2^k} = x^{(k)}.$$

□

Lemma 1.27 *If $x \in \mathbb{R}$, $m \in \mathbb{Z}$, and $k \in \mathbb{Z}$, then*

$$(2^k x)^{(m)} = 2^k x^{(k+m)}.$$

Proof $(2^k x)^{(m)} = 2^{-m} \lfloor 2^m (2^k x) \rfloor = 2^k (2^{-(k+m)} \lfloor 2^{k+m} x \rfloor = 2^k x^{(k+m)}$. □

Lemma 1.28 *If $x \in \mathbb{R}$, $m \in \mathbb{N}$, $n \in \mathbb{Z}$, then $n \leq x \Leftrightarrow n \leq x^{(m)}$.*

Proof $n \leq x \Leftrightarrow 2^m n \leq 2^m x \Leftrightarrow 2^m n \leq \lfloor 2^m x \rfloor \Leftrightarrow n \leq \frac{\lfloor 2^m x \rfloor}{2^m}$. □

Lemma 1.29 *If $x \in \mathbb{R}$, $m \in \mathbb{Z}$, and $-2^{-m} \leq x < 2^{-m}$, then*

$$x^{(m)} = \begin{cases} 0 & \text{if } x \geq 0 \\ -2^{-m} & \text{if } x < 0. \end{cases}$$

Proof Since $-1 \leq 2^m x < 1$, $-1 \leq \lfloor 2^m x \rfloor < 1$, which implies

$$\lfloor 2^m x \rfloor = \begin{cases} 0 \text{ if } x \geq 0 \\ -1 \text{ if } x < 0 \end{cases}$$

and

$$x^{(m)} = 2^{-m} \lfloor 2^m x \rfloor = \begin{cases} 0 & \text{if } x \geq 0 \\ -2^{-m} & \text{if } x < 0. \end{cases}$$

□

If $k > 0$, then $x^{(-k)} = 2^k \lfloor 2^{-k} x \rfloor$ is the largest multiple of 2^k that does not exceed x.

Example Let

$$x = \frac{51}{8} = 6 + \frac{3}{8}.$$

Then

$$x^{(-2)} = 2^2\lfloor 2^{-2}x\rfloor = 4\left\lfloor \frac{51}{32}\right\rfloor = 4.$$

The following two lemmas have found application in the analysis of floating-point adders. (See the proof of Lemma 17.10.)

Lemma 1.30 *If $k \in \mathbb{N}$, $n \in \mathbb{N}$, and $x \in \mathbb{R}$, then*

(a) $\left\lfloor \frac{x}{2^n}\right\rfloor^{(-k)} \leq \frac{x^{(-k)}}{2^n}$;
(b) $\frac{x^{(-k)}+2^k}{2^n} \leq \left\lfloor \frac{x}{2^n}\right\rfloor^{(-k)} + 2^k$.

Proof

(a) By Lemma 1.2,

$$\left\lfloor \frac{x}{2^n}\right\rfloor^{(-k)} = 2^k\left\lfloor \frac{\left\lfloor \frac{x}{2^n}\right\rfloor}{2^k}\right\rfloor = 2^k\left\lfloor \frac{x}{2^{k+n}}\right\rfloor = 2^k\left\lfloor \frac{\left\lfloor \frac{x}{2^k}\right\rfloor}{2^n}\right\rfloor \leq 2^k\frac{\left\lfloor \frac{x}{2^k}\right\rfloor}{2^n} = \frac{x^{(-k)}}{2^n}.$$

(b) By Lemmas 1.3, 1.11, and 1.2,

$$\begin{aligned}
\frac{x^{(-k)}+2^k}{2^n} &= \frac{2^k\left(\left\lfloor \frac{x}{2^k}\right\rfloor + 1\right)}{2^n} \\
&= 2^{k-n}\left(\left\lfloor \frac{x}{2^k}\right\rfloor + 1\right) \\
&= 2^{k-n}\left(2^n\left\lfloor \frac{\left\lfloor \frac{x}{2^k}\right\rfloor}{2^n}\right\rfloor + \left\lfloor \frac{x}{2^k}\right\rfloor \bmod 2^n + 1\right) \\
&\leq 2^{k-n}\left(2^n\left\lfloor \frac{\left\lfloor \frac{x}{2^k}\right\rfloor}{2^n}\right\rfloor + (2^n-1) + 1\right) \\
&= 2^k\left(\left\lfloor \frac{\left\lfloor \frac{x}{2^n}\right\rfloor}{2^k}\right\rfloor + 1\right) \\
&= \left\lfloor \frac{x}{2^n}\right\rfloor^{(-k)} + 2^k.
\end{aligned}$$

□

Lemma 1.31 *Let $k \in \mathbb{N}$, $n \in \mathbb{N}$, $x \in \mathbb{R}$, and $y \in \mathbb{R}$. $y = x^{(-k)}$. If $\lfloor 2^{-k}x \rfloor = \lfloor 2^{-k}y \rfloor$ and $2^{-k}x \notin \mathbb{Z}$, then*

$$\left(-\left\lfloor \frac{y}{2^n} \right\rfloor - 1\right)^{(-k)} = \left(-\frac{x}{2^n}\right)^{(-k)}.$$

Proof We simplify the expression on the left using Lemmas 1.2 and 1.5:

$$\begin{aligned}
\left(-\left\lfloor \frac{y}{2^n} \right\rfloor - 1\right)^{(-k)} &= 2^k \left\lfloor -\frac{\left\lfloor \frac{y}{2^n} \right\rfloor + 1}{2^k} \right\rfloor \\
&= -2^k \left(\left\lfloor \frac{\left\lfloor \frac{y}{2^n} \right\rfloor}{2^k} \right\rfloor + 1 \right) \\
&= -2^k \left(\left\lfloor \frac{\left\lfloor \frac{y}{2^k} \right\rfloor}{2^n} \right\rfloor + 1 \right) \\
&= -2^k \left(\left\lfloor \frac{\left\lfloor \frac{x}{2^k} \right\rfloor}{2^n} \right\rfloor + 1 \right) \\
&= -2^k \left(\left\lfloor \frac{x}{2^{k+n}} \right\rfloor + 1 \right)
\end{aligned}$$

For the expression on the right, we apply the same two lemmas and Lemma 1.4:

$$\begin{aligned}
\left(-\frac{x}{2^n}\right)^{(-k)} &= 2^k \left\lfloor -\frac{\frac{x}{2^n}}{2^k} \right\rfloor \\
&= 2^k \left\lfloor \frac{-x}{2^{k+n}} \right\rfloor \\
&= 2^k \left\lfloor \frac{\left\lfloor \frac{-x}{2^k} \right\rfloor}{2^n} \right\rfloor \\
&= 2^k \left\lfloor \frac{-\left\lfloor \frac{x}{2^k} \right\rfloor - 1}{2^n} \right\rfloor \\
&= -2^k \left(\left\lfloor \frac{\left\lfloor \frac{x}{2^k} \right\rfloor}{2^n} \right\rfloor + 1 \right) \\
&= -2^k \left(\left\lfloor \frac{x}{2^{k+n}} \right\rfloor + 1 \right) \\
&= \left(-\left\lfloor \frac{y}{2^k} \right\rfloor - 1\right)^{(-k)}.
\end{aligned}$$

□

Chapter 2
Bit Vectors

We shall use the term *bit vector* as a synonym of *integer*. Thus, a bit vector may be positive, negative, or zero. However, only a nonnegative bit vector may be associated with a *width*:

Definition 2.1 If $x \in \mathbb{N}$, $n \in \mathbb{N}$, and $x < 2^n$, then x is a bit vector of width n, or an n-bit vector.

Note that the width of a bit vector is not unique, since an n-bit vector is also an m-bit vector for all $m > n$.

The *bit slice* and *bit extraction* functions are defined as follows:

Definition 2.2 Let $x \in \mathbb{Z}$, $i \in \mathbb{Z}$, and $j \in \mathbb{Z}$.

(a) $x[i : j] = \lfloor (x \bmod 2^{i+1})/2^j \rfloor$;
(b) $x[i] = x[i : i]$.

Notation For the purpose of resolving ambiguous expressions, these operators take precedence over the basic arithmetic operators, e.g.,

$$xy[i : j][k : \ell] = x((y[i : j])[k : \ell]).$$

For any $x \in \mathbb{Z}$, the *binary representation* of x is $(\dots b_2 b_1 b_0)_2$, where $b_i = x[i]$ for all $i \in \mathbb{N}$. We may omit the subscript when the intention is clear. We shall show (Lemma 2.40) that distinct integers have distinct binary representations, so that we may write

$$x = (\dots b_2 b_1 b_0)_2.$$

In the sequel, we shall extend this notation to non-integral floating-point numbers: for $k \in \mathbb{N}$,

$$2^{-k}x = (\dots b_k . b_{k-1} \dots b_1 b_0)_2.$$

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_2

If x is an n-bit vector, then it is easily seen that $x[i] = 0$ for all $i \geq n$, and we may omit the leading zeroes in the representation of x:

$$x = (\ldots 000b_{n-1} \ldots b_1 b_0)_2 = (b_{n-1} \ldots b_1 b_0)_2.$$

We shall also show (Corollary 2.38) that in this case,

$$x = \sum_{k=0}^{n-1} 2^k x[k].$$

Since bit extraction is defined as a special case of bit slice, we shall discuss the latter in Sect. 2.1 and the former in Sect. 2.2. Section 2.3 deals with the basic RTL operation of *concatenation*.

Arithmetic hardware employs a variety of encoding schemes to represent integers and rational numbers as bit vectors. Floating-point representations are the subject of Chap. 5. In Sects. 2.4 and 2.5, we address the simpler integer and fixed-point formats.

2.1 Bit Slices

Lemma 2.1 *For all $x \in \mathbb{Z}$, $i \in \mathbb{N}$, and $j \in \mathbb{N}$, $x[i:j]$ is an $(i+1-j)$-bit vector.*

Proof By Lemmas 1.1 and 1.10, $x[i:j] \in \mathbb{N}$. By Lemma 1.11,

$$x[i:j] = \lfloor (x \bmod 2^{i+1})/2^j \rfloor \leq (x \bmod 2^{i+1})/2^j < 2^{i+1}/2^j = 2^{i+1-j}.$$

□

Example Let $x = 93 = (1011101)_2$. Then

$$x[4:2] = \lfloor (x \bmod 2^5)/2^2 \rfloor = \lfloor (93 \bmod 32)/4 \rfloor = \lfloor 29/4 \rfloor = 7 = (111)_2$$

is a 3-bit vector and

$$x[10:7] = \lfloor (93 \bmod 2^{11})/2^7 \rfloor = \lfloor 93/128 \rfloor = 0 = (0000)_2$$

is a 4-bit vector.

Lemma 2.2 *Let $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $i \in \mathbb{Z}$, and $j \in \mathbb{N}$. If $x \bmod 2^{i+1} = y \bmod 2^{i+1}$, then $x[i:j] = y[i:j]$.*

Proof $x[i:j] = \lfloor (x \bmod 2^{i+1})/2^j \rfloor = \lfloor (y \bmod 2^{i+1})/2^j \rfloor = y[i:j]$. □

Lemma 2.3 *For all $x \in \mathbb{Z}$ and $i \in \mathbb{Z}$,*

$$x[i:0] = x \bmod 2^{i+1}.$$

Proof $x[i:0] = \lfloor (x \bmod 2^{i+1})/2^0 \rfloor = \lfloor x \bmod 2^{i+1} \rfloor = x \bmod 2^{i+1}$. □

Lemma 2.4 *Let $x \in \mathbb{Z}$ and $i \in \mathbb{N}$. If $-2^{i+1} \leq x < 2^{i+1}$, then*

$$x[i:0] = \begin{cases} x & \text{if } x \geq 0 \\ x + 2^{i+1} & \text{if } x < 0. \end{cases}$$

Proof If $x \geq 0$, the claim follows from Lemma 2.3. If $-2^{i+1} \leq x < 0$, then by Lemmas 2.3, 1.15, and 1.11,

$$x[i:0] = x \bmod 2^{i+1} = (x + 2^{i+1}) \bmod 2^{i+1} = x + 2^{i+1}.$$

□

If $-2^j \leq x < 0$, then $x[i:j]$ is the bit vector of width $i - j + 1$ consisting of all 1s:

Lemma 2.5 *Let $x \in \mathbb{Z}$, $i \in \mathbb{N}$, and $j \in \mathbb{N}$. If $i \geq j$ and $-2^j \leq x < 0$, then $x[i:j] = 2^{i-j+1} - 1$.*

Proof By Lemmas 2.3 and 1.15, $x \bmod 2^{i+1} = x + 2^{i+1}$. Thus, by Definition 2.2, Lemma 1.1, and Definition 1.1,

$$x[i:j] = \lfloor (x + 2^{i+1})/2^j \rfloor = \lfloor x/2^j + 2^{i-j+1} \rfloor = \lfloor x/2^j \rfloor + 2^{i-j+1} = 2^{i-j+1} - 1.$$

□

Corollary 2.6 *If $i \in \mathbb{N}$, $j \in \mathbb{N}$, and $i \geq j$, then $(-1)[i:j] = 2^{i-j+1} - 1$.*

The following results are derived from corresponding properties of *mod*.

Lemma 2.7 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $i \in \mathbb{Z}$, $j \in \mathbb{Z}$, and $k \in \mathbb{Z}$, if $j \geq 0$ and $k \geq i$, then*

(a) $(x + y[k:0])[i:j] = (x + y)[i:j]$;
(b) $(x - y[k:0])[i:j] = (x + y)[i:j]$;
(c) $(xy[k:0])[i:j] = (xy)[i:j]$.

Proof By Definition 2.2 and Lemmas 1.22 and 1.19,

$$\begin{aligned}(x + y[i:0])[i:j] &= \lfloor (x + (y \bmod 2^{k+1}) \bmod 2^{i+1})/2^j \rfloor \\ &= \lfloor ((x + (y \bmod 2^{k+1}) \bmod 2^{k+1}) \bmod 2^{i+1})/2^j \rfloor \\ &= \lfloor (((x + y) \bmod 2^{k+1}) \bmod 2^{i+1})/2^j \rfloor\end{aligned}$$

$$= \lfloor ((x + y) \bmod 2^{i+1})/2^j \rfloor$$
$$= (x + y)[i : j].$$

The other claims follow similarly from Lemmas 1.23 and 1.24. □

By expanding the modulus, we may express a bit slice in terms of the floor alone:

Lemma 2.8 *Let $x \in \mathbb{Z}$, $i \in \mathbb{Z}$, and $j \in \mathbb{Z}$. If $i \geq j$, then*

$$x[i-1 : j] = \left\lfloor \frac{x}{2^j} \right\rfloor - 2^{i-j} \left\lfloor \frac{x}{2^i} \right\rfloor = \left\lfloor \frac{x}{2^j} \right\rfloor \bmod 2^{i-j}.$$

Proof Applying Definitions 2.2 and 1.3 and Lemma 1.1, we have

$$\begin{aligned} x[i-1 : j] &= \lfloor (x \bmod 2^i)/2^j \rfloor \\ &= \left\lfloor \frac{x - \lfloor x/2^i \rfloor 2^i}{2^j} \right\rfloor \\ &= \left\lfloor \frac{x}{2^j} - \left\lfloor \frac{x}{2^i} \right\rfloor 2^{i-j} \right\rfloor \\ &= \left\lfloor \frac{x}{2^j} \right\rfloor - 2^{i-j} \left\lfloor \frac{x}{2^i} \right\rfloor. \end{aligned}$$

The second claim follows from Definition 1.3 and Lemmas 2.8 and 1.2:

$$\left\lfloor \frac{x}{2^j} \right\rfloor \bmod 2^{i-j} = \left\lfloor \frac{x}{2^j} \right\rfloor - 2^{i-j} \left\lfloor \frac{\lfloor 2^{-j} x \rfloor}{2^{i-j}} \right\rfloor = \left\lfloor \frac{x}{2^j} \right\rfloor - 2^{i-j} \left\lfloor \frac{x}{2^i} \right\rfloor = x[i-1 : j].$$

□

In most cases of interest, the index arguments of $x[i : j]$ satisfy $i \geq j \geq 0$. However, the following lemma is worth noting.

Lemma 2.9 *For all $x \in \mathbb{Z}$, $i \in \mathbb{Z}$, and $j \in \mathbb{Z}$, if either $i < 0$ or $i < j$, then $x[i : j] = 0$.*

Proof Suppose $i < 0$. Since $-(i + 1) \geq 0$, $2^{-(i+1)} x \in \mathbb{Z}$. Applying Definition 1.3 and Lemma 1.1, we have

$$\begin{aligned} x \bmod 2^{i+1} &= x - \lfloor x/2^{i+1} \rfloor 2^{i+1} \rfloor \\ &= x - \lfloor 2^{-(i+1)} x \rfloor 2^{i+1} \rfloor \\ &= x - 2^{-(i+1)} x 2^{i+1} \rfloor \\ &= 0. \end{aligned}$$

If $i < j$, then by Lemma 1.11,

$$x \bmod 2^{i+1} < 2^{i+1} \leq 2^j,$$

and hence

$$x[i:j] = \lfloor (x \bmod 2^{i+1})/2^j \rfloor \leq (x \bmod 2^{i+1})/2^j < 1,$$

which, together with Lemma 2.1, implies $x[i:j] = 0$. □

Here is another case in which a bit slice may be reduced to 0:

Lemma 2.10 *Let $x \in \mathbb{N}$, $i \in \mathbb{N}$, and $j \in \mathbb{N}$. If x is a j-bit vector, then $x[i:j] = 0$.*

Proof By Lemmas 1.10 and 1.11,

$$0 \leq x \bmod 2^{i+1} \leq x < 2^j.$$

Therefore, by Lemma 1.1 and Definition 1.1,

$$0 \leq x[i:j] = \lfloor (x \bmod 2^{i+1})/2^j \rfloor \leq (x \bmod 2^{i+1})/2^j < 1,$$

which, together with Lemma 2.1, implies $x[i:j] = 0$. □

Corollary 2.11 *For all $i \in \mathbb{N}$ and $j \in \mathbb{N}$, $0[i:j] = 0$.*

A slice of a right-shifted bit vector, $\lfloor x/2^k \rfloor$, may always be represented as a slice of x:

Lemma 2.12 *For all $x \in \mathbb{N}$, $i \in \mathbb{N}$, $j \in \mathbb{N}$, and $k \in \mathbb{N}$,*

$$\lfloor x/2^k \rfloor[i:j] = x[i+k:j+k].$$

Proof Let $q = \lfloor x/2^{i+k+1} \rfloor$ and $r = x \bmod 2^{i+k+1}$, so that $x = 2^{i+k+1}q + r$ and $0 \leq r < 2^{i+k+1}$. Then

$$\lfloor x/2^k \rfloor = \lfloor 2^{i+1}q + r/2^k \rfloor = 2^{i+1}q + \lfloor r/2^k \rfloor,$$

where $\lfloor r/2^k \rfloor \leq r/2^k \leq 2^{i+1}$. Hence,

$$\lfloor x/2^k \rfloor \bmod 2^{i+1} = \lfloor r/2^k \rfloor$$

and by Definition 2.2 and Lemma 1.2,

$$\lfloor x/2^k \rfloor[i:j] = \lfloor \lfloor r/2^k \rfloor / 2^j \rfloor = \lfloor r/2^{k+j} \rfloor = \lfloor (x \bmod 2^{i+k+1})/2^{k+j} \rfloor = x[i:j].$$

□

Lemma 2.13 *For all $x \in \mathbb{N}$, $i \in \mathbb{N}$, and $k \in \mathbb{N}$,*

$$\lfloor x/2^k \rfloor[i:0] = \lfloor x[i+k:0]/2^k \rfloor.$$

Proof Applying Lemma 2.12, Definition 2.2, and Lemma 2.3 in succession, we have

$$\lfloor x/2^k \rfloor[i:0] = x[i+k:k] = \lfloor (x \bmod 2^{i+k+1})/2^k \rfloor = \lfloor x[i+k:0]/2^k \rfloor.$$

□

Lemma 2.14 *For all* $x \in \mathbb{N}$*,* $i \in \mathbb{N}$*,* $j \in \mathbb{N}$*, and* $k \in \mathbb{N}$*,*

$$(2^k x)[i:j] = x[i-k:j-k].$$

Proof If $k \leq i$, then by Definition 2.2 and Lemma 1.18,

$$\begin{aligned}(2^k x)[i:j] &= \lfloor (2^k x \bmod 2^{i+1})/2^j \rfloor \\ &= \lfloor 2^k (x \bmod 2^{i-k+1})/2^j \rfloor \\ &= \lfloor (x \bmod 2^{i-k+1})/2^{j-k} \rfloor \\ &= x[i-k:j-k].\end{aligned}$$

If $i < k$, then by Definition 2.2 and Corollary 1.12,

$$\begin{aligned}(2^k x)[i:j] &= \lfloor (2^k x \bmod 2^{i+1})/2^j \rfloor \\ &= \lfloor (2^{i+1}(2^{k-i-1}x) \bmod 2^{i+1})/2^j \rfloor \\ &= 0\end{aligned}$$

and $x[i-k:j-k] = 0$ by Lemma 2.9. □

The next lemma provides an alternate expression for a left-shifted bit slice with lower limit 0:

Lemma 2.15 *For all* $x \in \mathbb{N}$*,* $i \in \mathbb{N}$*, and* $k \in \mathbb{N}$*,*

$$2^k x[i:0] = (2^k x)[i+k:0].$$

Proof By Lemmas 1.18 and 2.3,

$$(2^k x)[i+k:0] = 2^k x \bmod 2^{i+k+1} = 2^k (x \bmod 2^{i+1}) = 2^k x[i:0].$$

□

We note two cases in which a bit slice of $x + 2^k y$ can be simplified.

Lemma 2.16 *Let* $x \in \mathbb{Z}$*,* $y \in \mathbb{Z}$*,* $m \in \mathbb{N}$*,* $n \in \mathbb{N}$*, and* $k \in \mathbb{N}$*. If* $k \leq m$ *and* $x < 2^k$*, then*

$$(x + 2^k y)[n:m] = y[n-k:m-k].$$

Proof By Lemma 2.9, we may assume that $n \geq m \geq k$. Since $0 \leq x/2^k < 1$,

$$\lfloor (x + 2^k y)/2^k \rfloor = \lfloor y + 2/2^k \rfloor = y.$$

We apply Lemma 2.12, substituting $x + 2^k y$ for x, $n - k$ for i, and $m - k$ for j:

$$y[n - k : m - k] = \lfloor (x + 2^k y)/2^k \rfloor [n - k : m - k] = (x + 2^k y)[n : m].$$

□

Lemma 2.17 *Let $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $m \in \mathbb{N}$, $n \in \mathbb{N}$, and $k \in \mathbb{N}$. If $n < k$, then*

$$(x + 2^k y)[n : m] = x[n : m].$$

Proof Since $2^k y = 2^{n+1}(2^{k-n-1} y)$, where $2^{k-n-1} y \in \mathbb{Z}$, Lemma 1.15 implies $(x + 2^k y) \bmod 2^{n+1} = x \bmod 2^{n+1}$. The lemma follows from Lemma 2.2. □

Here is an important lemma that decomposes a slice into two subslices.

Lemma 2.18 *Let $x \in \mathbb{Z}$, $m \in \mathbb{N}$, $n \in \mathbb{N}$, and $p \in \mathbb{N}$. If $m \leq p \leq n$, then*

$$x[n : m] = 2^{p-m} x[n : p] + x[p-1 : m].$$

Proof The proof consists of three applications of Lemma 2.8:

$$2^{p-m} x[n : p] = 2^{p-m} \left(\left\lfloor \frac{x}{2^p} \right\rfloor - 2^{n+1-p} \left\lfloor \frac{x}{2^{n+1}} \right\rfloor \right),$$

$$x[p-1 : m] = \left\lfloor \frac{x}{2^m} \right\rfloor - 2^{p-m} \left\lfloor \frac{x}{2^p} \right\rfloor,$$

and hence,

$$\begin{aligned} 2^{p-m} x[n : p] + x[p-1 : m] &= \left\lfloor \frac{x}{2^m} \right\rfloor - 2^{n+1-m} \left\lfloor \frac{x}{2^{n+1}} \right\rfloor \\ &= x[n : m]. \end{aligned}$$

□

Compositions of bit slices may be reduced by means of the following.

Lemma 2.19 *For all $x \in \mathbb{N}$, $i \in \mathbb{N}$, $j \in \mathbb{N}$, $k \in \mathbb{N}$, and $\ell \in \mathbb{N}$,*

$$x[i : j][k : l] = \begin{cases} x[k + j : \ell + j] & \text{if } k \leq i - j \\ x[i : \ell + j] & \text{if } k > i - j. \end{cases}$$

Proof By Lemma 2.12,

$$\begin{aligned} x[i:j][k:\ell] &= \lfloor x/2^j \rfloor[i-j:0][k:\ell] \\ &= (\lfloor x/2^j \rfloor \bmod 2^{i-j+1})[k:\ell] \\ &= (\lfloor(\lfloor x/2^j \rfloor \bmod 2^{i-j+1}) \bmod 2^{k+1})/2^{\ell}\rfloor. \end{aligned}$$

If $k \leq i - j$, then this reduces, by Corollary 1.20, to

$$\lfloor(\lfloor x/2^j \rfloor \bmod 2^{k+1})/2^{\ell}\rfloor = \lfloor x/2^j \rfloor[k:\ell] = x[k+j:\ell+j].$$

On the other hand, if $k > i - j$, then by Lemma 1.11,

$$\lfloor x/2^j \rfloor \bmod 2^{i-j+1} < 2^{i-j+1} < 2^{k+1},$$

and by Lemma 1.11, the expression reduces instead to

$$\lfloor(\lfloor x/2^j \rfloor \bmod 2^{i-j+1})/2^{\ell}\rfloor = \lfloor x/2^j \rfloor[i-j:\ell] = x[i:\ell+j].$$

□

2.2 Bit Extraction

Instead of Definition 2.2, we could have defined $x[n]$ more directly as follows.

Lemma 2.20 *For all $x \in \mathbb{Z}$ and $n \in \mathbb{Z}$,*

$$x[n] = \lfloor x/2^n \rfloor \bmod 2.$$

Proof By Lemmas 2.8 and 1.2 and Definition 1.3,

$$\begin{aligned} x[n] = x[(n+1)-1:n] &= \lfloor x/2^n \rfloor - 2\lfloor x/2^{n+1} \rfloor \\ &= \lfloor x/2^n \rfloor - 2\lfloor \lfloor x/2^n \rfloor/2 \rfloor \\ &= \lfloor x/2^n \rfloor \bmod 2. \end{aligned}$$

□

Corollary 2.21 *For all $x \in \mathbb{Z}$ and $n \in \mathbb{Z}$, $x[n] \in \{0, 1\}$.*

Here is an equivalent recursive definition that may be used in inductive proofs.

Lemma 2.22 *For all $x \in \mathbb{Z}$ and $n \in \mathbb{N}$,*

$$x[n] = \begin{cases} x \bmod 2 & \text{if } n = 0 \\ \lfloor x/2 \rfloor[n-1] & \text{if } n > 0. \end{cases}$$

Proof The base case is the $n = 0$ case of Lemma 2.20. The inductive case is an instance of Lemma 2.12, with $k = 1$ and $i = j = n - 1$. □

A number of important properties of bit extraction are special cases of the results of Sect. 2.1 We list some of them here without proof.

Lemma 2.23 *For all $x \in \mathbb{Z}$ and $n \in \mathbb{Z}$, if $n < 0$, then $x[n] = 0$.*

Lemma 2.24 *For all $k \in \mathbb{Z}$, $0[k] = 0$.*

Lemma 2.25 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $n \in \mathbb{Z}$, and $k \in \mathbb{Z}$, if $k < n$ and $x \bmod 2^n = y \bmod 2^n$, then*

$$x[k] = y[k].$$

Lemma 2.26 *For all $n \in \mathbb{Z}$, if x is an n-bit vector, then $x[n] = 0$.*

Lemma 2.27 *Let $x \in \mathbb{Z}$ and $n \in \mathbb{N}$. If $-2^n \leq x < 0$, then $x[n] = 1$.*

Corollary 2.28 *For all $i \in \mathbb{N}$, $(-1)[i] = 1$.*

Lemma 2.29 *For all $x \in \mathbb{Z}$, $n \in \mathbb{Z}$, and $k \in \mathbb{Z}$,*

$$(2^k x)[n + k] = x[n].$$

Lemma 2.30 *For all $x \in \mathbb{N}$, $i \in \mathbb{N}$, and $k \in \mathbb{N}$,*

$$\lfloor x/2^k \rfloor[i] = x[i + k].$$

Lemma 2.31 *For all $x \in \mathbb{Z}$, $i \in \mathbb{Z}$, $j \in \mathbb{Z}$, and $k \in \mathbb{Z}$, if $0 \leq k \leq i - j$, then*

$$x[i : j][k] = x[j + k].$$

Lemma 2.32 *For all $x \in \mathbb{Z}$, $m \in \mathbb{Z}$, and $n \in \mathbb{Z}$, if $m \leq n$, then*

$$x[n : m] = 2^{n-m} x[n] + x[n-1 : m].$$

Lemma 2.33 *For all $x \in \mathbb{Z}$, $m \in \mathbb{Z}$, and $n \in \mathbb{Z}$, if $m \leq n$, then*

$$x[n : m] = x[m] + 2x[n : m+1].$$

Lemma 2.34 *Let $n \in \mathbb{N}$ and $k \in \mathbb{N}$, and let x be an n-bit vector. If $k < n$ and $x \geq 2^n - 2^k$, then $x[k] = 1$.*

Proof Since $2^n - 2^k \leq x < 2^n$, $2^{n-k} - 1 \leq x/2^k < 2^{n-k}$, and by Definition 1.1, $\lfloor x/2^k \rfloor = 2^{n-k} - 1$. Now by Lemma 2.20, $x[k] = (2^{n-k} - 1) \bmod 2 = 1$. □

Corollary 2.35 *For all $n \in \mathbb{Z}$ and $x \in \mathbb{N}$, if $2^n \leq x < 2^{n+1}$, then $x[n] = 1$.*

Lemma 2.36 *For all $n \in \mathbb{N}$ and $i \in \mathbb{Z}$, $(2^n)[i] = 1 \Leftrightarrow i = n$.*

Proof By Lemma 2.20, $(2^n)[n] = \lfloor 2^n/2^n \rfloor \bmod 2 = 1 \bmod 2 = 1$.

Suppose $i \neq n$. If $i < n$, then 2^i is an n-bit vector and Lemma 2.26 applies. If $i > n$, then

$$(2^i)[n] = \lfloor 2^i/2^n \rfloor \bmod 2 = mod 2^{i-n} \bmod 2 = 0.$$

□

The following lemma and its corollary justify the notation discussed at the beginning of this chapter.

Lemma 2.37 *For all $x \in \mathbb{N}$, $i \in \mathbb{N}$, and $j \in \mathbb{N}$,*

$$\sum_{k=j}^{i} 2^{k-j} x[k] = x[i:j].$$

Proof If $i < j$, then both sides of the equation reduce to 0 by Lemma 2.9. We proceed by induction. Thus, for $i \geq j$, applying Lemma 2.32, we have

$$\begin{aligned}\sum_{k=j}^{i} 2^{k-j} x[k] &= 2^{i-j} x[i] + \sum_{k=j}^{i-1} 2^{k-j} x[k] \\ &= 2^{i-j} x[i] + x[i-i:j] \\ &= x[i:j].\end{aligned}$$

□

Corollary 2.38 *If $n \in \mathbb{N}$, $n > 0$, and x is an n-bit vector, then*

$$\sum_{k=0}^{n-1} 2^k x[k] = x.$$

Proof This follows from Lemmas 2.37 and 2.4. □

The next lemma allows us to define a bit vector in a natural way as a sequence of bits. That is, given a sequence of 1-bit vectors $b_0, \ldots, b_{n-1}$, we may say, without ambiguity, *Let x be the bit vector of width n defined by $x[k] = b_k$ for $k = 0, \ldots, n-1$* .... The existence of such a bit vector is guaranteed by Lemma 2.39; its uniqueness is ensured by Corollary 2.38.

Lemma 2.39 *Let $x = \sum_{i=0}^{n-1} 2^i b_i$, where $n \in \mathbb{N}$ and $b_i \in \{0, 1\}$, $i = 0, \ldots, n-1$. Then for $k = 0, \ldots, n-1$, $x[k] = b_k$.*

Proof Let $U = \sum_{i=k+1}^{n-1} 2^i b_i$ and $L = \sum_{i=0}^{k-1} 2^i b_i$. Then

$$x = U + 2^k b_k + L.$$

Since

$$U = 2^{k+1} \sum_{i=k+1}^{n-1} 2^{i-(k+1)} b_i,$$

$$x \bmod 2^{k+1} = U + 2^k b_k + L \bmod 2^{k+1} = 2^k b_k + L \bmod 2^{k+1}$$

by Lemma 1.15. But since

$$L \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

and

$$2^k b_k + L \leq 2^k + 2^k - 1 < 2^{k+1},$$

$$2^k b_k + L \bmod 2^{k+1} = 2^k b_k + L$$

by Lemma 1.11. Thus, by Definitions 2.2 and 2.2,

$$x[k] = \lfloor (x \bmod 2^{k+1})/2^k \rfloor = \lfloor (2^k b_k + L)/2^k \rfloor = \lfloor b_k + L/2^k \rfloor = b_k.$$

□

A bit vector is uniquely determined by its binary representation:

Lemma 2.40 *Let $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$. If $x[k] = y[k]$ for all $k \in \mathbb{N}$, then $x = y$.*

Proof The proof is by induction on $|x| + |y|$.

Suppose $x \neq y$. We must show that for some $k \in \mathbb{N}$, $x[k] \neq y[k]$. We may assume that $x[0] = y[0]$, and hence $\lfloor x/2 \rfloor \neq \lfloor y/2 \rfloor$, for otherwise

$$x = 2\lfloor x/2 \rfloor + x[0] = 2\lfloor y/2 \rfloor + y[0] = y.$$

Since $x \neq y$ and $x[0] = y[0]$, at least one of x and y must be different from both 0 and -1, and hence, by Lemma 1.3,

$$|\lfloor x/2 \rfloor| + |\lfloor y/2 \rfloor)| < |x| + |y|.$$

By induction, there exists $k \in \mathbb{N}$ such that $\lfloor x/2 \rfloor[k] \neq \lfloor y/2 \rfloor[k]$, and consequently, by Lemma 2.30,

$$x[k+1] = \lfloor x/2 \rfloor[k] \neq \lfloor y/2 \rfloor[k] = y[k+1].$$

□

2.3 Concatenation

If $x = (\beta_{m-1} \cdots \beta_0)_2$ and $y = (\gamma_{n-1} \cdots \gamma_0)_2$ are considered as bit vectors of widths m and n, respectively, then the concatenation of x and y is the $(m+n)$-bit vector

$$(\beta_{m-1} \cdots \beta_0 \gamma_{n-1} \cdots \gamma_0)_2.$$

This notion is extended by the following function, which takes a list of bit vectors and widths, coerces each bit vector to its associated width, and concatenates the results:

Definition 2.3 For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$,

$$cat(x, m, y, n) = 2^n x[m-1:0] + y[n-1:0].$$

This construction is extended recursively to $2k$ arguments for arbitrary $k \in \mathbb{Z}^+$:

$$cat(x_1, n_1, x_2, n_2, \ldots, x_k, n_k) = cat(x_1, n_1, cat(x_2, n_2, \ldots, x_k, n_k), n_2 + \ldots + n_k),$$

where $x_i \in \mathbb{Z}$ and $n_i \in \mathbb{N}$ for $i = 1, \ldots, k$.

Associativity follows immediately:

Lemma 2.41 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $z \in \mathbb{Z}$, $m \in \mathbb{N}$, $n \in \mathbb{N}$, and $p \in \mathbb{N}$,*

$$cat(cat(x, m, y, n), z, p) = cat(x, m, y, n, z, p).$$

Lemma 2.42 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$, $cat(x, m, y, n)$ is an $(m+n)$-bit vector.*

Proof By Lemma 2.1, $x[m-1:0] < 2^m$ and $y[n-1:0] < 2^n$. It follows that $x[m-1:0] \leq 2^m - 1$ and $y[n-1:0] \leq 2^n - 1$, and hence,

$$\begin{aligned} cat(x, m, y, n) &= 2^n x[m-1:0] + y[n-1:0] \\ &\leq 2^n(2^m - 1) + (2^n - 1) \\ &= 2^{n+m} - 1 \\ &< 2^{n+m}. \end{aligned}$$

□

We note several trivial cases:

Lemma 2.43 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$,*

$$cat(x, m, y, 0) = x[m-1:0]$$

and

$$cat(x, 0, y, n) = cat(0, m, y, n) = y[n-1 : 0].$$

Proof These are simple consequences of Definition 2.3 and Lemmas 2.9 and 2.11. □

Notation In standard RTL syntax, the concatenation of two bit vectors ϕ and ψ is denoted by $\{\phi, \psi\}$. This notation depends on a characteristic shared by conventional hardware description languages: any expression that represents a bit vector has an associated (explicit or implicit) width. For example, the expression `sig[3:0]` is understood to be of width 4, and the expression `5'b01001` identifies the constant 9 as a bit vector of width 5. We shall incorporate this construct into our informal mathematical notation through an abuse of Verilog syntax, representing $cat(x, m, y, n)$ as

$$\{m'x, n'y\}.$$

The width specifier may be omitted in a context in which it can be inferred by default.

Example If $x \in \{0, 1\}$ and y has been identified as a bit vector of width n, then

$$\{x, y, z[i : j], w[k]\} = cat(x, 1, y, n, z[i : j], i + 1 - j, w[k], 1).$$

The following is a restatement of Lemma 2.18:

Lemma 2.44 *Let $x \in \mathbb{Z}$, $m \in \mathbb{N}$, $n \in \mathbb{N}$, and $p \in \mathbb{N}$. If $m \le p \le n$, then*

$$x[n : m] = \{x[n : p], x[p-1 : m]\}.$$

Corollary 2.45 *Let $x \in \mathbb{Z}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$. If $m \le n$, then*

$$x[n : m] = \{x[n], x[n-1 : m]\} = \{x[n : m+1], x[m]\}.$$

Lemma 2.46 *Let $z = \{m'x, n'y\}$, where $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$. Then*

$$z[n-1 : 0] = y[n-1 : 0]$$

and

$$z[n + m-1 : n] = x[m-1 : 0].$$

Proof By Definition 2.3, we have

$$z = 2^n x[m-1 : 0] + y[n-1 : 0],$$

where $0 \leq y[n-1:0] < 2^n$ by Lemma 2.1. Thus, by Lemmas 2.3, 1.15, and 1.11,

$$z[n-1:0] = z \bmod 2^n = y[n-1:0] \bmod 2^n = y[n-1:0].$$

Now by Definition 2,

$$z[n+m-1:n] = \lfloor (z \bmod 2^{n+m})/2^n \rfloor.$$

But Lemma 2.42 yields $z < 2^{n+m}$ and hence, by Lemma 1.11,

$$z[n+m-1:n] = \lfloor z/2^n \rfloor = \lfloor x[m-1:0] + y[n-1:0]/2^n \rfloor.$$

Finally, by Lemma 1.1, this reduces to

$$x[m-1:0] + \lfloor y[n-1:0]/2^n \rfloor = x[m-1:0].$$

□

Corollary 2.47 *Let $x_1 \in \mathbb{Z}$, $y_1 \in \mathbb{Z}$, $x_2 \in \mathbb{Z}$, $y_2 \in \mathbb{Z}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$. If*

$$\{m\,'\,x_1, n\,'\,y_1\} = \{m\,'\,x_2, n\,'\,y_2\},$$

then $x_1[m-1:0] = x_2[m-1:0]_2$ and $y_1[n-1:0]_1 = y_2[n-1:0]$.

Lemma 2.48 *Let $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $m \in \mathbb{N}$, $n \in \mathbb{N}$, $i \in \mathbb{N}$, and $j \in \mathbb{N}$. If $i \geq j$, then*

$$\{m\,'\,x, n\,'\,y\}[i:j] = \begin{cases} y[i:j] & \text{if } n > i \\ x[i-n:j-n] & \text{if } m+n > i \geq j \geq n \\ x[m-1:j-n] & \text{if } i \geq m+n \text{ and } j \geq n \\ \{x[i-n:0], y[n-1:j]\} & \text{if } m+n > i \geq n > j \\ \{x[m-1:0], y[n-1:j]\} & \text{if } i \geq n+m \text{ and } n > j. \end{cases}$$

Proof Let $z = \{x[m-1:0], y[n-1:0]\}$. By Lemma 2.46,

$$y[n-1:0] = z[n-1:0]$$

and

$$x[m-1:0] = z[n+m-1:n]$$

and by Lemma 2.42, z is an $(m+n)$-bit vector. Our goal is to compute $z[i:j]$. We consider five cases as suggested by the lemma statement, each of which involves two or more applications of Lemma 2.19.
Case 1: $n > i$

By Lemma 2.19,

$$z[i:j] = z[n{-}1:0][i:j] = y[n{-}1:0][i:j] = y[i:j].$$

Case 2: $m+n > i \geq j \geq n$
By Lemma 2.19,

$$\begin{aligned} z[i:j] &= z[m+n{-}1:n][i-n:j-n] \\ &= x[m{-}1:0][i-n:j-n] \\ &= x[i-n:j-n]. \end{aligned}$$

Case 3: $i \geq m+n$ and $j \geq n$
By Lemma 2.44,

$$z[i:j] = \{z[i:m+n], z[m+n{-}1:j]\}.$$

But $z[i:m+n] = 0$ by Lemma 2.10 and hence

$$z[i:j] = z[m+n{-}1:j]$$

by Lemma 2.43. Now by Lemma 2.19,

$$\begin{aligned} z[m+n{-}1:j] &= z[m+n{-}1:n][m{-}1:j-n] \\ &= x[m{-}1:0][m{-}1:j-n] \\ &= x[m{-}1:j-n]. \end{aligned}$$

Case 4: $m+n > i \geq n > j$
By Lemma 2.44,

$$z[i:j] = \{z[i:n], z[n{-}1:j]\}.$$

But by Lemma 2.19,

$$\begin{aligned} z[i:n] &= z[m+n{-}1:n][i-n:0] \\ &= x[m{-}1:0][i-n:0] \\ &= x[i-n:0] \end{aligned}$$

and

$$\begin{aligned} z[n{-}1:j] &= z[n{-}1:0][n{-}1:j] \\ &= y[n{-}1:0][n{-}1:j] \\ &= y[n{-}1:j]. \end{aligned}$$

Case 5: $i \geq n+m$ and $n > j$

By Lemma 2.44,

$$z[i:j] = \{z[i:m+n], z[m+n-1:n], z[n-1:j]\}.$$

As in Case 4, $z[n-1:j] = y[n-1:j]$. By Lemma 2.10, $z[i:m+n] = 0$, and hence by Lemma 2.43,

$$z[i:j] = \{z[m+n-1:n], z[n-1:j]\} = \{x[m-1:0], y[n-1:j]\}.$$

□

Corollary 2.49 *If $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $m \in \mathbb{N}$, $n \in \mathbb{N}$, and $i \in \mathbb{N}$, then*

$$\{m\,'\,x, n\,'\,y\}[i] = \begin{cases} y[i] & \text{if } i < n \\ x[i-n] & \text{if } n \leq i < m+n \\ 0 & \text{if } n+m \leq i. \end{cases}$$

Proof The cases listed correspond to the first three cases of Lemma 2.48 with $i = j$. Note that for the third case, the lemma gives $x[m-1:i-n]$, but since $i > n+m$, i.e., $i-n > m-1$, this reduces to 0 by Lemma 2.9. □

2.4 Integer Formats

The simplest of all bit vector encoding schemes is the *unsigned integer* format, whereby the first 2^n natural numbers, i.e., the bit vectors of width n, are represented by themselves under the identity mapping. However trivial, it will be convenient to have an explicit definition of this correspondence:

Definition 2.4 If r is a n-bit vector, where $n \in \mathbb{N}$, then

$$ui(r, n) = r.$$

Somewhat more interesting is the *signed integer* format, which maps the set of 2^n integers x in the range $-2^{n-1} \leq x < 2^{n-1}$ to the set of bit vectors of width n and may be defined by

$$x \longmapsto x[n-1:0].$$

With respect to this mapping, the most significant bit of the encoding of x,

$$x[n-1:0][n-1] = x[n-1],$$

is 0 if $0 \leq x < 2^{n-1}$ (by Lemma 2.26) and 1 if $-2^{n-1} \leq x < 0$ (by Lemma 2.27), and is therefore considered the *sign bit* of the encoding.

The integer represented by a given encoding is computed by the following function, as affirmed by Lemma 2.50 below:

Definition 2.5 If r is a n-bit vector, where $n \in \mathbb{N}$, then

$$si(r, n) = \begin{cases} r - 2^n & \text{if } r[n-1] = 1 \\ r & \text{if } r[n-1] = 0. \end{cases}$$

Lemma 2.50 *Let $n \in \mathbb{N}$ and $x \in \mathbb{Z}$. If $-2^{n-1} \leq x < 2^{n-1}$, then*

$$si(x[n-1:0], n) = x.$$

Proof If $0 \leq x < 2^{n-1}$, then $x[n-1:0] = x$ by Lemma 2.4, and $x[n-1] = 0$ by Lemma 2.26. Thus,

$$si(x[n-1:0], n) = si(x, n) = x.$$

If $-2^{n-1} \leq x < 0$, then $x[n-1:0] = x+2^n$ by Lemma 2.4, and $(x+2^n)[n-1] = 1$ by Corollary 2.35. Thus,

$$si(x[n-1:0], n) = si(x + 2^n, n) = (x + 2^n) - 2^n = x.$$

□

This scheme is also known as the n-bit *two's complement* encoding, because if $0 \leq x < 2^n$, then the encoding of $-x$ is the complement of x with respect to 2^n, i.e.,

$$x + (-x)[n-1:0] = x + (-x + 2^n) = 2^n.$$

Lemma 2.51 *If $n \in \mathbb{N}$, $r \in \mathbb{N}$, $i \in \mathbb{N}$, and $j \in \mathbb{N}$ with $j \leq i < n$, then*

$$si(r, n)[i:j] = r[i:j].$$

Lemma 2.52 *If $n \in \mathbb{N}$, $k \in \mathbb{N}$, and r is an n-bit vector, then*

$$si(2^k r, k + n) = 2^k si(r, n).$$

Proof This follows easily from Definition 2.5 and Lemma 2.29. □

An n-bit integer encoding is converted to an m-bit encoding, where $m > n$, by *sign extension*:

Definition 2.6 Let r be an n-bit vector, where $n \in \mathbb{N}$, and let $m \in \mathbb{N}$, $m \geq n$. Then

$$sextend(m, n, r) = si(r, n)[m-1:0].$$

A sign extension of an integer encoding r represents the same value as r:

Lemma 2.53 *Let r be an n-bit vector, where $n \in \mathbb{N}$, and let $m \in \mathbb{N}$, $m \geq n$. Then*

$$si(sextend(m, n, r), m) = si(r, n).$$

Proof First suppose $r[n-1] = 0$. Then $si(r, n) = r$ and by Corollary 2.35, $0 \leq r < 2^{n-1}$. By Lemma 2.4,

$$sextend(m, n, r) = si(r, n)[m-1 : 0] = r[m-1 : 0] = r,$$

and since Lemma 2.26 implies $r[m-1] = 0$,

$$si(sextend(m, n, x), m) = si(r, m) = r = si(r, n).$$

Now suppose $r[n - 1] = 1$. Then by Lemma 2.26, $2^{n-1} \leq r < 2^n$. Now $si(r, n) = r - 2^n$, where $-2^{m-1} \leq -2^{n-1} \leq r - 2^n < 0$. By Lemma 2.4,

$$sextend(m, n, r) = si(r, n)[m-1 : 0] = (r - 2^n)[m-1 : 0] = r - 2^n + 2^m.$$

But since $2^{m-1} \leq r-2^n+2^m < 2^m$. Corollary 2.35 implies $(r-2^n+2^m)[m-1] = 1$, and hence

$$si(sextend(m, n, r), m) = si(r-2^n+2^m, m) = r-2^n+2^m-2^m-r-2^n = si(rl, n).$$

□

Given an approximation Y of an integer X, the following lemma provides a condition under which the n-bit signed integer represented by Y is an equally accurate approximation of the n-bit signed integer represented by X. This result is useful in approximating the signed integer values of a "redundant" representation, i.e., a representation of an integer as a sum or difference of two vectors. (See, for example, the proof of Lemma 18.5.)

Lemma 2.54 *Let $X \in \mathbb{Z}$, $Y \in \mathbb{Z}$, and $n \in \mathbb{Z}$, with $n > 0$. If*

$$|si(X \bmod 2^n, n)| + |X - Y| < 2^{n-1},$$

then

$$si(X \bmod 2^n, n) - si(Y \bmod 2^n, n) = X - Y.$$

Proof Let $\bar{X} = X \bmod 2^n$, $\bar{Y} = Y \bmod 2^n$, and $k = |X - Y|$.
Case 1: $\lfloor \frac{X}{2^n} \rfloor = \lfloor \frac{Y}{2^n} \rfloor$.

In this case, $\bar{X} - \bar{Y} = X - Y$.

Suppose $\bar{X} \leq \bar{Y}$. If $\bar{X} \geq 2^{n-1}$, then $\bar{Y} \geq 2^{n-1}$ and

$$si(\bar{X}, n) - si(\bar{Y}, n) = (\bar{X} - 2^n) - (\bar{Y} - 2^n) = \bar{X} - \bar{Y} = X - Y,$$

but if $\bar{X} < 2^{n-1}$, then $\bar{X} = si(\bar{X}, n) < 2^{n-1} - k$, which implies $\bar{Y} < 2^{n-1}$ and

$$si(\bar{X}, n) - si(\bar{Y}, n) = \bar{X} - \bar{Y} = X - Y.$$

On the other hand, suppose $\bar{X} > \bar{Y}$. If $\bar{X} < 2^{n-1}$, then $\bar{Y} < 2^{n-1}$ and

$$si(\bar{X}, n) - si(\bar{Y}, n) = \bar{X} - \bar{Y} = X - Y,$$

but if $\bar{X} \geq 2^{n-1}$, then $si(\bar{X}, n) = \bar{X} - 2^n$, and since $si(\bar{X}, n) > -2^{n-1} + k$, $\bar{X} > 2^{n-1} + k$, which implies $\bar{Y} > 2^{n-1}$ and

$$si(\bar{X}, n) - si(\bar{Y}, n) = (\bar{X} - 2^n) - (\bar{Y} - 2^n) = \bar{X} - \bar{Y} = X - Y.$$

Case 2: $\lfloor \frac{X}{2^n} \rfloor \neq \lfloor \frac{Y}{2^n} \rfloor$.

Suppose $X < Y$. Let $m = \lfloor \frac{X}{2^n} \rfloor$. Then

$$2^n m \leq X < 2^n(m+1) \leq Y < X + 2^{n-1} < 2^n(m+2).$$

Thus, $\bar{X} = X - 2^n m$ and

$$\bar{Y} = Y - 2^n(m+1) = k + X - 2^n(m+1) = k + (\bar{X} + 2^n m) - 2^n(m+1) = k - 2^n + \bar{X} < k < 2^{n-1}.$$

But then

$$\bar{X} = \bar{Y} + 2^n - k \geq 2^n - k > 2^{n-1}$$

and

$$si(\bar{X}, n) - si(\bar{Y}, n) = \bar{X} - 2^n - \bar{Y} = (\bar{Y} + 2^n - k) - 2^n - \bar{Y} = -k = X - Y.$$

The case $X > Y$ is similar. □

2.5 Fixed-Point Formats

A fixed-point format may be thought of as derived from an integer format by inserting an implicit binary point following some specified number of leading bits. The rational value represented by an n-bit vector r with respect to an unsigned or signed fixed-point format of width n with m integer bits is computed as follows:

Definition 2.7 Let $n \in \mathbb{Z}$ and $m \in \mathbb{Z}$ with $n > 0$ and let r be a bit vector of width n.

(a) $uf(r, n, m) = 2^{m-n} ui(r) = 2^{m-n} r$;

(b) $sf(r, n, m) = 2^{m-n} si(r, n) = \begin{cases} 2^{m-n} r & \text{if } r < 2^{n-1} \\ 2^{m-n} r - 2^m & \text{if } r \geq 2^{n-1}, \end{cases}$.

The number of fractional bits of a fixed-point format of width n and m integer bits is $f = n - m$. Note that while n must be positive, there is no restriction on m. If $m > n$, then the interpreted value is an integer with $m - n$ trailing zeroes and $f < 0$; if $m < 0$, then the interpreted value is a fraction with $-m$ leading zeroes and $f > n$.

We have the following expression for a bit slice of an encoding in terms of the encoded value:

Lemma 2.55 *Let $n \in \mathbb{N}$, $m \in \mathbb{N}$, $i \in \mathbb{N}$, and $j \in \mathbb{N}$ with $m \leq n$ and $j \leq i < n$. Let $f = n - m$. Let r be an n-bit vector and suppose that either $x = uf(r, n, m)$ or $x = sf(r, n, m)$. Then*

$$r[i : j] = 2^{f-j} \left(x^{(f-j)} - x^{(f-i-1)} \right).$$

Proof If $x = uf(r, n, m)$, then $r = 2^f x$; if $x = sf(r, n, m)$, then either $r = 2^f x$ or $r = 2^f x + 2^n$. In any case, by Lemmas 2.2 and 2.8,

$$r[i : j] = (2^f x)[i : j] = \left\lfloor \frac{2^f x}{2^j} \right\rfloor = \left\lfloor \frac{2^f x}{2^{1+i}} \right\rfloor = 2^{f-j} \left(x^{(f-j)} - x^{(f-i-1)} \right).$$

□

Corollary 2.56 *Let $n \in N$ and $m \in N$ with $m \leq n$ and let $f = n - m$. Let $k \in \mathbb{Z}$ with $f - n \leq k < f$. Let r be an n-bit vector and suppose that either $x = uf(x, n, m)$ or $x = sf(x, n, m)$. Then*

$$x^{(k)} = x \Leftrightarrow r[f - k - 1 : 0] = 0.$$

Proof By Lemma 2.55,

$$r[f - k - 1 : 0] = 2^f \left(x^{(f)} - x^{(k)} \right) = 2^f \left(x - x^{(k)} \right).$$

□

The following result is useful in determining the value of a fixed-point encoding:

Lemma 2.57 *Let $n \in N$ and $m \in N$ with $m \leq n$. Let r be an n-bit vector and $x = sf(r, n, m)$. If $y \in \mathbb{Z}$ satisfies $r \equiv y \pmod{2^n}$ and $-2^{n-1} \leq y < 2^{n-1}$, then*

$$x = 2^{m-n} y.$$

Proof Since $r \equiv y \pmod{2^n}$ and $0 \leq r < 2^n$, $r = y \bmod 2^n = y[n-1:0]$. By Lemma 2.50, $y = si(r, n)$ and hence

$$x = sf(r, n, m) = 2^{m-n} si(r, n) = 2^{m-n} y.$$

□

Chapter 3
Logical Operations

In this chapter, we define and analyze the four basic logical operations: the unary "not", or complement, and the binary "and", "inclusive or" and "exclusive or". These are commonly known as *bit-wise* operations, as each one may be computed by performing a certain operation on each bit of its argument (in the unary case) or each pair of corresponding bits of its arguments (for binary operations). For example, the logical "and" $x \mathbin{\&} y$ of two bit vectors may be specified in a bit-wise manner as the bit vector z such that for all $k \in \mathbb{N}$, $z[k] = 1$ iff $x[k] = y[k] = 1$.

In the context of our formalization, however, the logical operations are more naturally defined as arithmetic functions: the complement is constructed as an arithmetic difference and the binary operations are defined by recursive formulas, which facilitate inductive proofs of their relevant properties. Among these are the bit-wise characterizations, as represented by Lemmas 3.7 and 3.20.

3.1 Binary Operations

Following standard RTL syntax, we denote "and", "inclusive or" and "exclusive or" with the infix symbols &, |, and ^, respectively.

Definition 3.1 For all $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$,

(a) $$x \mathbin{\&} y = \begin{cases} 0 & \text{if } x = 0 \text{ or } y = 0 \\ x & \text{if } x = y \\ 2 \cdot (\lfloor x/2 \rfloor \mathbin{\&} \lfloor y/2 \rfloor) + (x \bmod 2) \mathbin{\&} (y \bmod 2) & \text{otherwise,} \end{cases}$$

(b) $$x \mid y = \begin{cases} y & \text{if } x = 0 \text{ or } x = y \\ x & \text{if } y = 0 \\ 2 \cdot (\lfloor x/2 \rfloor \mid \lfloor y/2 \rfloor) + (x \bmod 2) \mid (y \bmod 2) & \text{otherwise;} \end{cases}$$

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_3

(c) $x \mathbin{\hat{}} y = \begin{cases} y & \text{if } x = 0 \\ x & \text{if } y = 0 \\ 0 & \text{if } x = y \\ 2 \cdot (\lfloor x/2 \rfloor \mathbin{\hat{}} \lfloor y/2 \rfloor) + (x \bmod 2) \mathbin{\hat{}} (y \bmod 2) & \text{otherwise.} \end{cases}$

It is not obvious that these are admissible recursive definitions, i.e., that each of them is satisfied by a unique function. To establish this, it suffices to demonstrate the existence of a *measure* function $\mu : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ that strictly decreases on each recursive call. Thus, we define

$$\mu(x, y) = \begin{cases} 0 & \text{if } x = y \\ |xy| & \text{if } x \neq y. \end{cases}$$

For the admissibility of each of the three definitions, we must show that μ satisfies the following two inequalities, corresponding to the two recursive calls, under the restrictions $x \neq 0$, $y \neq 0$, and $x \neq y$:

(1) $\mu(\lfloor x/2 \rfloor, \lfloor y/2 \rfloor) < \mu(x, y)$.
(2) $\mu(x \bmod 2, y \bmod 2) < \mu(x, y)$..

Since the restrictions imply that at least one of x and y is neither 0 nor -1, (1) follows from Lemma 1.3. To establish (2), note that either $x \bmod 2 = 0$, $y \bmod 2 = 0$, or $x \bmod 2 = 1 = y \bmod 2$. In any case,

$$\mu(x \bmod 2, y \bmod 2) = 0 < |xy| = \mu(x, y).$$

The proof of the following is a typical inductive argument based on the recursion of Definition 3.1 (a).

Lemma 3.1 *If* $x \in \mathbb{N}$ *and* $y \in \mathbb{Z}$, *then* $0 \leq x \mathbin{\&} y \leq x$.

Proof We may assume that $x \neq 0$, $y \neq 0$, and $x \neq y$. Thus,

$$x \mathbin{\&} y = 2(\lfloor x/2 \rfloor \mathbin{\&} \lfloor y/2 \rfloor) + x \bmod 2 \mathbin{\&} y \bmod 2$$

and by induction,

$$0 \leq x \mathbin{\&} y \leq 2\lfloor x/2 \rfloor + x \bmod 2 = x.$$

□

Corollary 3.2 *If* x *is an* n*-bit vector and* $y \in \mathbb{Z}$, *then* $x \mathbin{\&} y$ *is an* n*-bit vector.*

Proof By Lemma 3.1, $0 \leq x \mathbin{\&} y \leq x < 2^n$ □

Lemma 3.3 *If* x *and* y *and* n*-bit vectors, then so are* $x \mid y$ *and* $x \mathbin{\hat{}} y$.

Proof The same argument applies to both operations. The claim is trivial if $n = 0$, $x = 0$, $y = 0$, or $x = y$. In all other cases, $\lfloor x/2 \rfloor$ and $\lfloor y/2 \rfloor$ are $(n-1)$-bit vectors

and by induction, so is, for example, $\lfloor x/2 \rfloor \mid \lfloor y/2 \rfloor$. Thus,

$$x \mid y = 2(\lfloor x/2 \rfloor \mid \lfloor y/2 \rfloor) + (x \bmod 2) \mid (y \bmod 2) \leq 2 \cdot (2^{n-1} - 1) + 1 < 2^n.$$

□

Lemma 3.4 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, and $n \in \mathbb{N}$,*

(a) $(x \,\&\, y) \bmod 2^n = (x \bmod 2^n) \,\&\, (y \bmod 2^n)$;
(b) $(x \mid y) \bmod 2^n = (x \bmod 2^n) \mid (y \bmod 2^n)$;
(c) $(x \,\hat{}\, y) \bmod 2^n = (x \bmod 2^n) \,\hat{}\, (y \bmod 2^n)$.

Proof We present the proof for (a); (b) and (c) are similar.

We may assume that $n > 0$, $x \neq 0$, $y \neq 0$, and $x \neq y$. By Definition 3.1 (a) and Lemma 1.22,

$$\begin{aligned}
&(x \,\&\, y) \bmod 2^n \\
&= (2 \cdot (\lfloor x/2 \rfloor \,\&\, \lfloor y/2 \rfloor) + (x \bmod 2) \,\&\, (y \bmod 2)) \bmod 2^n \\
&= \big((2 \cdot (\lfloor x/2 \rfloor \,\&\, \lfloor y/2 \rfloor)) \bmod 2^n + (x \bmod 2) \,\&\, (y \bmod 2)\big) \bmod 2^n.
\end{aligned}$$

By induction and Lemmas 1.18, 2.3, and 2.13, the first addend may be written as

$$\begin{aligned}
(2 \cdot (\lfloor x/2 \rfloor \,\&\, \lfloor y/2 \rfloor)) \bmod 2^n &= 2 \cdot \Big((\lfloor x/2 \rfloor \,\&\, \lfloor y/2 \rfloor) \bmod 2^{n-1}\Big) \\
&= 2 \cdot \Big((\lfloor x/2 \rfloor \bmod 2^{n-1}) \,\&\, (\lfloor y/2 \rfloor \bmod 2^{n-1})\Big) \\
&= 2 \cdot (\lfloor x/2 \rfloor[n-2:0] \,\&\, \lfloor y/2 \rfloor[n-2:0]) \\
&= 2 \cdot (\lfloor x[n-1:0]/2 \rfloor \,\&\, \lfloor y[n-1:0]/2 \rfloor)\,,
\end{aligned}$$

and by Lemmas 2.22 and 2.31, the second addend is

$$\begin{aligned}
(x \bmod 2) \,\&\, (y \bmod 2) &= x[0] \,\&\, y[0] = x[n-1:0][0] \,\&\, y[n-1:0][0] \\
&= (x[n-1:0] \bmod 2) \,\&\, (y[n-1:0] \bmod 2).
\end{aligned}$$

Thus, by Definition 3.1 (a) and Lemmas 3.2 and 2.3,

$$\begin{aligned}
(x \,\&\, y) \bmod 2^n &= (x[n-1:0] \,\&\, y[n-1:0]) \bmod 2^n \\
&= x[n-1:0] \,\&\, y[n-1:0] \\
&= (x \bmod 2^n) \,\&\, (y \bmod 2^n).
\end{aligned}$$

□

Lemma 3.5 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, and $n \in \mathbb{N}$,*

(a) $\lfloor (x \mathbin{\&} y)/2^n \rfloor = \lfloor x/2^n \rfloor \mathbin{\&} \lfloor y/2^n \rfloor$;
(b) $\lfloor (x \mid y)/2^n \rfloor = \lfloor x/2^n \rfloor \mid \lfloor y/2^n \rfloor$;
(c) $\lfloor (x \mathbin{\char`\^} y)/2^n \rfloor = \lfloor x/2^n \rfloor \mathbin{\char`\^} \lfloor y/2^n \rfloor$.

Proof We present the proof for (a); (b) and (c) are similar.

We may assume that $n > 0$, $x \neq 0$, $y \neq 0$, and $x \neq y$. By Lemma 1.2, induction, and Definition 3.1 (a),

$$\begin{aligned}
\lfloor (x \mathbin{\&} y)/2^n \rfloor &= \left\lfloor \lfloor (x \mathbin{\&} y)/2^{n-1} \rfloor /2 \right\rfloor \\
&= \left\lfloor (\lfloor x/2^{n-1} \rfloor \mathbin{\&} \lfloor y/2^{n-1} \rfloor)/2 \right\rfloor \\
&= \left\lfloor \lfloor x/2^{n-1} \rfloor /2 \rfloor \mathbin{\&} \lfloor x/2^{n-1} \rfloor /2 \rfloor \right\rfloor \\
&= \left\lfloor x/2^n \rfloor \mathbin{\&} \lfloor y/2^n \right\rfloor .
\end{aligned}$$

□

All three binary logical operators commute with the bit slice operator:

Lemma 3.6 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $i \in \mathbb{N}$, and $j \in \mathbb{N}$,*

(a) $(x \mathbin{\&} y)[i:j] = x[i:j] \mathbin{\&} y[i:j]$;
(b) $(x \mid y)[i:j] = x[i:j] \mid y[i:j]$;
(c) $(x \mathbin{\char`\^} y)[i:j] = x[i:j] \mathbin{\char`\^} y[i:j]$.

Proof We present the proof for (a); (b) and (c) are similar.

We may assume that $n > 0$, $x \neq 0$, $y \neq 0$, and $x \neq y$. By Definition 2.2 and Lemmas 3.4 and 3.5,

$$\begin{aligned}
(x \mathbin{\&} y)[i:j] &= \left\lfloor ((x \mathbin{\&} y) \bmod 2^{i+1})/2^j \right\rfloor \\
&= \left\lfloor ((x \bmod 2^{i+1}) \mathbin{\&} (y \bmod 2^{i+1}))/2^j \right\rfloor \\
&= \lfloor (x \bmod 2^{i+1})/2^j \rfloor \mathbin{\&} \lfloor (y \bmod 2^{i+1})/2^j \rfloor \\
&= x[i:j] \mathbin{\&} y[i:j].
\end{aligned}$$

□

Corollary 3.7 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, and $k \in \mathbb{N}$,*

(a) $(x \mathbin{\&} y)[n] = x[n] \mathbin{\&} y[n]$;
(b) $(x \mid y)[n] = x[n] \mid y[n]$;
(c) $(x \mathbin{\char`\^} y)[n] = x[n] \mathbin{\char`\^} y[n]$.

Lemma 3.8 *For all* $x_1 \in \mathbb{Z}$, $y_1 \in \mathbb{Z}$, $x_2 \in \mathbb{Z}$, $y_2 \in \mathbb{Z}$, $m \in \mathbb{N}$, *and* $n \in \mathbb{N}$,

(a) $\{m'x_1, n'y_1\} \mathbin{\&} \{m'x_2, n'y_2\} = \{m'(x_1 \mathbin{\&} x_2), n'(y_1 \mathbin{\&} y_2)\}$;
(b) $\{m'x_1, n'y_1\} \mid \{m'x_2, n'y_2\} = \{m'(x_1 \mid x_2), n'(y_1 \mid y_2)\}$;
(c) $\{m'x_1, n'y_1\} \mathbin{\hat{}} \{m'x_2, n'y_2\} = \{m'(x_1 \mathbin{\hat{}} x_2), n'(y_1 \mathbin{\hat{}} y_2)\}$.

Proof We present the proof for (a); (b) and (c) are similar.

Let $C = \{m'x_1, n'y_1\} \mathbin{\&} \{m'x_2, n'y_2\}$. By Lemmas 3.4 and 2.48,

$$\begin{aligned}
&C \bmod 2^n \\
&\quad = \{x_1[m{-}1:0], y_1[n{-}1:0]\}[n{-}1:0] \mathbin{\&} \{x_2[m{-}1:0], y_2[n{-}1:0]\}[n{-}1:0] \\
&\quad = y_1[n{-}1:0] \mathbin{\&} y_2[n{-}1:0].
\end{aligned}$$

By Lemma 3.5,

$$\lfloor C/2^n \rfloor = \lfloor \{x_1[m{-}1:0], y_1[n{-}1:0]\}/2^n \rfloor \mathbin{\&} \lfloor \{x_2[m{-}1:0], y_2[n{-}1:0]\}/2^n \rfloor,$$

where, by Definition 2.3 and the properties of the floor,

$$\begin{aligned}
\lfloor \{x_i[m{-}1:0], y_i[n{-}1:0]\}/2^n \rfloor &= \lfloor (2^n x_i[m{-}1:0] + y_i[n{-}1:0])/2^n \rfloor \\
&= x_i[m{-}1:0] + \lfloor y_i[n{-}1:0]/2^n \rfloor \\
&= x_i[m{-}1:0].
\end{aligned}$$

Thus,

$$\lfloor C/2^n \rfloor = x_1[m{-}1:0] \mathbin{\&} x_2[m{-}1:0].$$

Finally, by Definitions 1.3 and 2.3,

$$\begin{aligned}
C &= \lfloor C/2^n \rfloor 2^n + (C \bmod 2^n)) \\
&= 2^n(x_1[m{-}1:0] \mathbin{\&} x_2[m{-}1:0]) + y_1[m{-}1:0] \mathbin{\&} y_2[m{-}1:0]) \\
&= \{x_1[m{-}1:0] \mathbin{\&} x_2[m{-}1:0], y_1[n{-}1:0] \mathbin{\&} y_2[n{-}1:0]\}.
\end{aligned}$$

□

Lemma 3.9 *For all* $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, *and* $n \in \mathbb{N}$,

(a) $2^n(x \mathbin{\&} y) = 2^n x \mathbin{\&} 2^n y$;
(b) $2^n(x \mid y) = 2^n x \mid 2^n y$;
(c) $2^n(x \mathbin{\hat{}} y) = 2^n x \mathbin{\hat{}} 2^n y$.

Proof We present the proof for (a); (b) and (c) are similar.

We may assume that $n > 0$, $x \neq 0$, $y \neq 0$, and $x \neq y$. By induction and Definition 3.1 (a),

$$
\begin{aligned}
2^n(x \mathbin{\&} y) &= 2\left(2^{n-1}(x \mathbin{\&} y)\right) \\
&= 2\left(2^{n-1}x \mathbin{\&} 2^{n-1}y\right) \\
&= 2\left(\lfloor 2^n x/2 \rfloor \mathbin{\&} \lfloor 2^n x/2 \rfloor\right) + (2^n x \bmod 2) \mathbin{\&} (2^n y \bmod 2) \\
&= 2^n x \mathbin{\&} 2^n y.
\end{aligned}
$$

□

Lemma 3.10 *For all $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, and $n \in \mathbb{N}$,*

(a) $2^n x \mathbin{\&} y = 2^n(x \mathbin{\&} \lfloor y/2^n \rfloor)$;
(b) $2^n x \mid y = 2^n(x \mid \lfloor y/2^n \rfloor) + y \bmod 2^n$;
(c) $2^n x \mathbin{\char`^} y = 2^n(x \mathbin{\char`^} \lfloor y/2^n \rfloor) + y \bmod 2^n$.

Proof

(a) The claim is trivial if $x = 0$, $y = 0$, or $y = 2^n x$; otherwise, by Definition 3.1 (a), induction, and Lemma 1.2,

$$
\begin{aligned}
2^n x \mathbin{\&} y &= 2\left(\lfloor 2^n x/2 \rfloor \mathbin{\&} \lfloor y/2 \rfloor\right) + (2^n x \bmod 2) \mathbin{\&} (y \bmod 2) \\
&= 2(2^{n-1}x \mathbin{\&} \lfloor y/2 \rfloor) + 0 \\
&= 2\left(2^{n-1}(x \mathbin{\&} \left\lfloor \lfloor y/2 \rfloor / 2^{n-1} \right\rfloor)\right) \\
&= 2^n(x \mathbin{\&} \lfloor y/2^n \rfloor).
\end{aligned}
$$

(b) Similarly,

$$
\begin{aligned}
2^n x \mid y &= 2\left(\lfloor 2^n x/2 \rfloor \mid \lfloor y/2 \rfloor\right) + (2^n x \bmod 2) \mid (y \bmod 2) \\
&= 2(2^{n-1}x \mid \lfloor y/2 \rfloor) + y \bmod 2 \\
&= 2\left(2^{n-1}\left(x \mid \left\lfloor \lfloor y/2 \rfloor / 2^{n-1} \right\rfloor\right) + \lfloor y/2 \rfloor \bmod 2^{n-1}\right) + y \bmod 2 \\
&= 2^n(x \mid \lfloor y/2^n \rfloor) + 2(\lfloor y/2 \rfloor \bmod 2^{n-1}) + y \bmod 2,
\end{aligned}
$$

where, by Lemmas 2.3 and 2.12,

$$
\begin{aligned}
2(\lfloor y/2 \rfloor \bmod 2^{n-1}) + y \bmod 2 &= 2\lfloor y/2 \rfloor[n-2:0] + y[0] \\
&= 2y[n{-}1:1] + y[0] \\
&= y[n{-}1:0] \\
&= y \bmod 2^n.
\end{aligned}
$$

The proof of (c) is similar to that of (b). □

Corollary 3.11 *Let $x \in \mathbb{Z}$ and let y be an n-bit vector, where $n \in \mathbb{N}$. Then*

$$2^n x \mid y = 2^n x + y.$$

Proof By Lemmas 3.10 and 1.11 and Definition 3.1 (b),

$$2^n x \mid y = 2^n(x \mid \lfloor y/2^n \rfloor) + y \bmod 2^n = 2^n(x \mid 0) + y = 2^n x + y.$$

□

Lemma 3.12 *For all $x \in \mathbb{Z}$ and $n \in \mathbb{N}$, $2^n \mid x = \begin{cases} x & \text{if } x[n] = 1 \\ x + 2^n & \text{if } x[n] = 0. \end{cases}$*

Proof By Definition 2.2 and Lemmas 3.4, 3.8, and 2.32,

$$\begin{aligned}
(2^n \mid x) \bmod 2^{n+1} &= (2^n \mid x)[n:0] \\
&= (2^n)[n:0] \mid x[n:0] \\
&= \{1'1, 0'(n-1)\} \mid \{x[n], x[n{-}1:0]\} \\
&= \{1'1, x[n{-}1:0]\} \\
&= \begin{cases} x[n:0] & \text{if } x[n] = 1 \\ x[n:0] + 2^n & \text{if } x[n] = 0 \end{cases} \\
&= \begin{cases} x \bmod 2^{n+1} & \text{if } x[n] = 1 \\ x \bmod 2^{n+1} + 2^n & \text{if } x[n] = 0. \end{cases}
\end{aligned}$$

By Lemma 3.5,

$$\lfloor (2^n \mid x)/2^{n+1} \rfloor = 0 \mid \lfloor x/2^{n+1} \rfloor = \lfloor x/2^{n+1} \rfloor.$$

The lemma follows from Definition 1.3. □

The logical "and" operator may be used to extract a bit slice:

Lemma 3.13 *Let $x \in \mathbb{Z}$, $n \in \mathbb{N}$, and $k \in \mathbb{N}$. If $k < n$, then*

$$x \mathbin{\&} (2^n - 2^k) = 2^k x[n{-}1:k].$$

Proof The proof is by induction on n. If $n = 1$, then $k = 0$ and

$$x \mathbin{\&} (2^n - 2^k) = x \mathbin{\&} 1 = 2(\lfloor x/2 \rfloor \mathbin{\&} 0) + (x \bmod 2) \mathbin{\&} 1 = 0 + x[0] = x[n{-}1:k].$$

If $n > 1$ and $k = 0$, then by induction and Lemmas 2.12 and 2.32,

$$
\begin{aligned}
x \mathbin{\&} (2^n - 2^k) = x \mathbin{\&} (2^n - 1)
&= 2(\lfloor x/2 \rfloor \mathbin{\&} \lfloor (2^n - 1)/2 \rfloor) + (x \bmod 2) \mathbin{\&} ((2^n - 1) \bmod 2) \\
&= 2(\lfloor x/2 \rfloor \mathbin{\&} (2^{n-1} - 1)) + (x \bmod 2) \mathbin{\&} 1 \\
&= 2\lfloor x/2 \rfloor[n - 2 : 0] + x[0] \\
&= 2x[n{-}1 : 1] + x[0] \\
&= x[n{-}1 : 0].
\end{aligned}
$$

In the remaining case, $n > k > 1$ and

$$
\begin{aligned}
x \mathbin{\&} (2^n - 2^k) &= 2(\lfloor x/2 \rfloor \mathbin{\&} \lfloor (2^n - 1)/2 \rfloor) + (x \bmod 2) \mathbin{\&} ((2^n - 2^k) \bmod 2) \\
&= 2(\lfloor x/2 \rfloor \mathbin{\&} (2^{n-1} - 2^{k-1})) + (x \bmod 2) \mathbin{\&} 0 \\
&= 2\lfloor x/2 \rfloor[n - 2 : k{-}1] \\
&= x[n{-}1 : k].
\end{aligned}
$$

□

Corollary 3.14 *For all $x \in \mathbb{Z}$ and $n \in \mathbb{N}$, $x \mathbin{\&} 2^n = 2^n x[n]$.*

Proof By Lemma 3.13,

$$x \mathbin{\&} 2^n = x \mathbin{\&} (2^{n+1} - 2^n) = x[n : n] = x[n].$$

□

3.2 Complement

We have a simple arithmetic definition of the logical complement.

Definition 3.2 For all $x \in \mathbb{Z}$, $\sim x = -x - 1$.

Lemma 3.15 *For all $x \in \mathbb{Z}$, $\sim(\sim x) = x$.*

Proof By Definition 3.2, $\sim(\sim x) = -(-x - 1) - 1 = x$. □

Lemma 3.16 *If $x \in \mathbb{Z}$ and $k \in \mathbb{N}$, then $\sim(2^k x) = 2^k(\sim x) + 2^k - 1$.*

Proof By Definition 3.2,

$$2^k(\sim x) + 2^k - 1 = 2^k(-x - 1) + 2^k - 1 = -2^k x - 1 = \sim(2^k x).$$

□

Lemma 3.17 *If* $x \in \mathbb{Z}$, $n \in \mathbb{N}$, *and* $n > 0$, *then* $\sim\lfloor x/n \rfloor = \lfloor \sim x/n \rfloor$.

Proof By Definition 3.2 and Lemma 1.5,

$$\lfloor (\sim x)/n \rfloor = \left\lfloor \frac{-x-1}{n} \right\rfloor = \left\lfloor -\frac{x+1}{n} \right\rfloor = -\left\lfloor \frac{x}{n} \right\rfloor - 1 = \sim\lfloor x/n \rfloor.$$

□

Lemma 3.18 *If* $x \in \mathbb{Z}$ *and* $n \in \mathbb{N}$, *then*

$$\sim x \bmod 2^n = 2^n - (x \bmod 2^n) - 1.$$

Proof First note that by Lemmas 1.10 and 1.11,

$$0 \leq 2^n - (x \bmod 2^n) - 1 < 2^n.$$

Therefore, by Lemmas 1.15, 1.23, and 1.11,

$$\begin{aligned} \sim x \bmod 2^n &= (-x-1) \bmod 2^n \\ &= (2^n - (x \bmod 2^n) - 1) \bmod 2^n \\ &= 2^n - (x \bmod 2^n) - 1. \end{aligned}$$

□

Notation For the purpose of resolving ambiguous expressions, the complement has higher precedence than the bit slice operator, e.g., $\sim x[i:j] = (\sim x)[i:j]$.

Lemma 3.19 *If* $x \in \mathbb{Z}$, $i \in \mathbb{N}$, $j \in \mathbb{N}$, *and* $j \leq i$, *then*

$$\sim x[i:j] = 2^{i+1-j} - x[i:j] - 1.$$

Proof By Definitions 3.2 and 2.2 and Lemmas 3.18, 1.1, and 1.5,

$$\begin{aligned} \sim x[i:j] &= \left\lfloor (\sim x \bmod 2^{i+1})/2^j \right\rfloor \\ &= \left\lfloor \frac{2^{i+1} - (x \bmod 2^{i+1}) - 1}{2^j} \right\rfloor \\ &= \frac{2^{i+1}}{2^j} + \left\lfloor -\frac{(x \bmod 2^{i+1}) + 1}{2^j} \right\rfloor \\ &= 2^{i+1-j} - \left\lfloor \frac{x \bmod 2^{i+1}}{2^j} \right\rfloor - 1 \\ &= 2^{i+1-j} - x[i:j] - 1. \end{aligned}$$

□

The usual bit-wise characterization of the complement is a special case of Lemma 3.19:

Corollary 3.20 *If $x \in \mathbb{Z}$ and $n \in \mathbb{N}$, then* $\sim x[n] \neq x[n]$.

Proof By Lemma 3.19, $\sim x[n] = 2^{n+1-n} - x[n] - 1 = 1 - x[n]$. □

The remaining results of this section are properties of complements of bit slices that have proved useful in manipulating expressions derived from RTL designs.

Lemma 3.21 *Let $x \in \mathbb{Z}$, $i \in \mathbb{N}$, $j \in \mathbb{N}$, $k \in \mathbb{N}$, and $\ell \in \mathbb{N}$. If $\ell \leq k \leq i - j$, then*

$$\sim(x[i:j])[k:\ell] = \sim x[k+j:\ell+j].$$

Proof By Lemmas 3.19 and 2.19,

$$\begin{aligned}\sim(x[i:j])[k:\ell] &= 2^{k+1-\ell} - x[i:j][k:\ell] - 1\\ &= 2^{(k+j)+1-(\ell+j)} - x[k+j:\ell+j] - 1\\ &= \sim x[k+j:\ell+j].\end{aligned}$$

□

Lemma 3.22 *If $x \in \mathbb{Z}$ and y in an n-bit vector, where $n \in \mathbb{N}$, then*

$$\sim(x[n-1:0]) \,\&\, y = \sim x[n-1:0] \,\&\, y.$$

Proof By Lemma 3.21, $\sim(x[n-1:0])[n-1:0] = \sim x[n-1:0]$, and hence by Lemmas 3.2, 2.4, and 3.6

$$\begin{aligned}\sim(x[n-1:0]) \,\&\, y &= (\sim(x[n-1:0]) \,\&\, y)\,[n-1:0]\\ &= \sim(x[n-1:0])[n-1:0] \,\&\, y[n-1:0]\\ &= \sim x[n-1:0] \,\&\, y.\end{aligned}$$

□

Lemma 3.23 *Let $x \in \mathbb{Z}$, $i \in \mathbb{N}$, $j \in \mathbb{N}$, $k \in \mathbb{N}$, and $\ell \in \mathbb{N}$. If $\ell \leq k \leq i - j$, then*

$$\sim(\sim x[i:j])[k:\ell] = x[k+j:\ell+j].$$

Proof By Lemmas 3.19, 2.19, and 3.15,

$$\begin{aligned}\sim(\sim x[i:j])[k:\ell] &= 2^{k+1-\ell} - \sim x[i:j][k:\ell] - 1\\ &= 2^{(k+j)+1-(\ell+j)} - \sim x[k+j:\ell+j] - 1\\ &= \sim(\sim x)[k+j:\ell+j]\\ &= x[k+j:\ell+j].\end{aligned}$$

□

3.3 Algebraic Properties

We conclude this chapter with a set of identities pertaining to special cases and compositions of logical operations.

The first two lemmas are immediate consequences of the definitions.

Lemma 3.24 *For all* $x \in \mathbb{Z}$,

(a) $x \mathbin{\&} 0 = 0$;
(b) $x \mid 0 = x$;
(c) $x \mathbin{\char`^} 0 = x$.

Lemma 3.25 *For all* $x \in \mathbb{Z}$ *and* $y \in \mathbb{Z}$,

(a) $x \mathbin{\&} x = x$;
(b) $x \mid x = x$;
(c) $x \mathbin{\char`^} x = 0$.

All of the remaining results of this section may be derived in a straightforward manner from Lemmas 3.20, 3.7, and 2.40.

Lemma 3.26 *For all* $x \in \mathbb{Z}$,

(a) $x \mathbin{\&} -1 = x$;
(b) $x \mid -1 = -1$;
(c) $x \mathbin{\char`^} -1 = {\sim}x$.

Lemma 3.27 *For all* $x \in \mathbb{Z}$ *and* $y \in \mathbb{Z}$,

(a) $x \mid y = 0 \Leftrightarrow x = y = 0$;
(b) $x \mathbin{\char`^} y = 0 \Leftrightarrow x = y$.

Proof Suppose $x \mid y = 0$. By Lemma 3.7, for all $k \in \mathbb{N}$

$$x[k] \mid y[k] = (x \mid y)[k] = 0[k] = 0,$$

and it is readily seen by exhaustive computation that this implies $x[k] = y[k] = 0$. It follows from Lemma 2.40 that $x = y = 0$. A similar argument applies to (b). □

The proofs of the remaining lemmas are sufficiently similar to that of Lemma 3.27 that they may be safely omitted.

Lemma 3.28 *For all* $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, *and* $n \in \mathbb{Z}$,

(a) $x \mathbin{\&} y = y \mathbin{\&} x$;
(b) $x \mid y = y \mid x$;
(c) $x \mathbin{\char`^} y = y \mathbin{\char`^} x$.

Lemma 3.29 *For all* $x \in \mathbb{N}$, $y \in \mathbb{N}$, *and* $z \in \mathbb{N}$,

(a) $(x \mathbin{\&} y) \mathbin{\&} z = x \mathbin{\&} (y \mathbin{\&} z)$;
(b) $(x \mid y) \mid z = x \mid (y \mid z)$;
(c) $(x \mathbin{\char`^} y) \mathbin{\char`^} z = x \mathbin{\char`^} (y \mathbin{\char`^} z)$.

Lemma 3.30 *For all* $x \in \mathbb{N}$, $y \in \mathbb{N}$, *and* $z \in \mathbb{N}$,

(a) $(x \mid y) \mathbin{\&} z = (x \mid y) \mathbin{\&} (x \mid z)$;
(b) $x \mathbin{\&} (y \mid z) = x \mathbin{\&} y \mid x \mathbin{\&} z$;
(c) $x \mathbin{\&} y \mid x \mathbin{\&} z \mid y \mathbin{\&} z = x \mathbin{\&} y \mid (x \mathbin{\hat{}} y) \mathbin{\&} z$.

Lemma 3.31 *For all* $x \in \mathbb{N}$ *and* $y \in \mathbb{N}$,

(a) $x \mathbin{\hat{}} y = x \mathbin{\&} {\sim}y \mid y \mathbin{\&} {\sim}x$;
(b) ${\sim}(x \mathbin{\hat{}} y) = ({\sim}x) \mathbin{\hat{}} y$.

Part II
Floating-Point Arithmetic

According to the IEEE floating-point standard [9], each of the elementary arithmetic operations of addition, multiplication, division, square root extraction, and fused multiplication-addition

> ...shall be performed as if it first produced an intermediate result correct to infinite precision and then rounded that result according to one of the [supported] modes ...

Since the operands (or operand, in the case of square root) and final result are represented as bit vectors, the relationship between inputs and outputs that is implicit in this specification, which is known as the *principle of correct rounding*, is as diagrammed in Fig. II.1.

That is, to compute the prescribed output, the pure mathematical operation to be implemented is applied to the decoded values of the inputs, and the result of this operation is rounded and encoded as the output. Of course, an implementation is not actually required to generate the output by such a procedure, and in fact, in

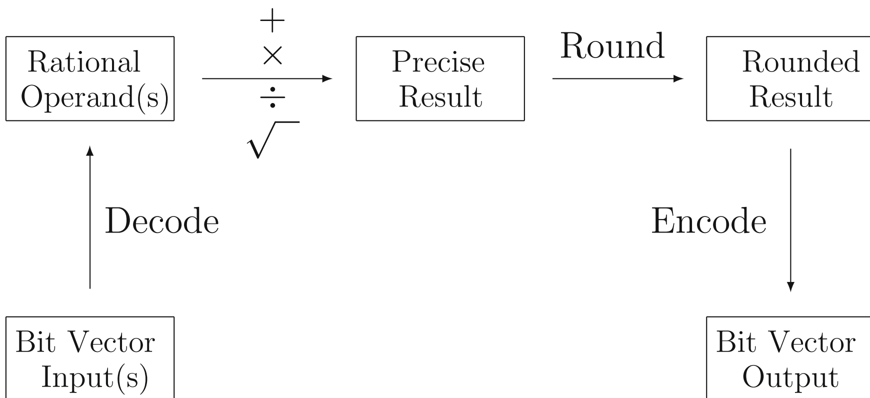


Fig. II.1 Principle of correct rounding

most cases an explicit representation of the precise unrounded result is infeasible. In Chaps. 4 and 5, we describe the common schemes for encoding real numbers as bit vectors. Chapter 6 addresses the problem of rounding.

As discussed in the preface, the formalization of the principle of correct rounding in the ACL2 logic, which is limited to rational arithmetic, presents a challenge in the case of the square root operation. In Chap. 7, we discuss our solution to this technical problem, which is based on a function that computes a rational number $\sqrt[(66)]{x}$, which approximates the real square root of a rational number x, and has the property that it rounds to the same value as does the precise square root with respect to any IEEE rounding mode and any format of interest. This chapter may be omitted by the reader who is not interested in the problem of formalization.

Chapter 4
Floating-Point Numbers

The designation *floating-point number* is a relative term, used to refer to a rational number that is representable with respect to a particular format, as described in Chap. 5. In this chapter, we discuss properties of real numbers that are relevant to these representations.

4.1 Decomposition

Floating-point arithmetic is based on the observation that every nonzero $x \in \mathbb{R}$ admits a unique representation of the form

$$x = \pm m \cdot 2^e,$$

where e is an integer, called the *exponent* of x, and m is a number in the range $1 \leq m < 2$, called the *significand* of x. These components are defined as follows.

Definition 4.1 Let $x \in \mathbb{R}$. If $x \neq 0$, then

(a) $sgn(x) = \frac{x}{|x|} = \begin{cases} 1 \text{ if } x > 0 \\ -1 \text{ if } x < 0; \end{cases}$

(b) $expo(x)$ is the unique integer that satisfies $2^{expo(x)} \leq |x| < 2^{expo(x)+1}$;

(c) $sig(x) = |x|2^{-expo(x)}$.

If $x = 0$, then $sgn(x) = expo(x) = sig(x) = 0$.

The decomposition property is immediate:

Lemma 4.1 *For all* $x \in \mathbb{R}$, $x = sgn(x)sig(x)2^{expo(x)}$.

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_4

Lemma 4.2 *For all $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}$,*

(a) If $2^n \le x$, then $n \le expo(x)$;
(b) If $0 < |x| < 2^{n+1}$, then $expo(x) \le n$;
(c) If $0 < |x| \le |y|$, then $expo(x) \le expo(y)$;
(d) $expo(2^n) = n$.

Proof

(a) If $n > expo(x)$, then $n \ge expo(x) + 1$, which implies $|x| < 2^{expo(x)+1} \le 2^n$.
(b) If $n < expo(x)$, then $n + 1 \le expo(x)$, which implies $2^{n+1} \le 2^{expo(x)} \le |x|$.
(c) Since $|x| \le |y| < 2^{expo(y)+1}$, this follows from (b).
(d) This is an immediate consequence of Definition 4.1. □

Lemma 4.3 *Let $x \in \mathbb{R}$.*

(a) If $x \neq 0$, then $1 \le sig(x) < 2$;
(b) If $1 \le x < 2$, then $sig(x) = x$;
(c) $sig(sig(x)) = sig(x)$.

Proof

(a) Definition 4.1 yields

$$1 = 2^{expo(x)}/2^{expo(x)} \le |x|/2^{expo(x)} < 2^{expo(x)+1}/2^{expo(x)} = 2.$$

(b) Since $2^0 \le |x| = x < 2^1$, $expo(x) = 0$, and hence $sig(x) = x/2^0 = x$.
(c) follows from (a) and (b). □

Lemma 4.4 *Let $x \in \mathbb{R}$. If $|x| = 2^e m$, where $e \in \mathbb{Z}$, $m \in \mathbb{R}$, and $1 \le m < 2$, then $m = sig(x)$ and $e = expo(x)$.*

Proof Since $1 \le m < 2$, $2^e \le 2^e m < 2^e 2 = 2^{e+1}$, where $2^e m = |x|$. It follows from Definition 4.1 that $e = expo(x)$, and therefore $sig(x) = |x|/2^e = m$. □

Changing the sign of a number does not affect its exponent or significand.

Lemma 4.5 *For all $x \in \mathbb{R}$,*

(a) $sgn(-x) = -sgn(x)$;
(b) $expo(-x) = expo(x)$;
(c) $sig(-x) = sig(x)$.

A shift does not affect the sign or significand.

Lemma 4.6 *If $x \in \mathbb{R}$, $x \neq 0$, and $n \in \mathbb{Z}$, then*

(a) $sgn(2^n x) = sgn(x)$;
(b) $expo(2^n x) = expo(x) + n$;
(c) $sig(2^n x) = sig(x)$.

Proof

(a) $sgn(2^n x) = 2^n x/|2^n x| = x/|x| = sgn(x)$.
(b) $2^{expo(x)} \le |x| < 2^{expo(x)+1} \Rightarrow 2^{expo(x)+n} \le |2^n x| < 2^{expo(x)+n+1}$.
(c) $sig(2^n x) = |2^n x| 2^{-(expo(x)+n)} = |x| 2^{-expo(x)} = sig(x)$. □

We have the following formulas for the components of a product.

Lemma 4.7 *Let $x \in \mathbb{R}$ and $y \in \mathbb{R}$. If $xy \neq 0$, then*

(a) $sgn(xy) = sgn(x)sgn(y)$;
(b) $expo(xy) = \begin{cases} expo(x) + expo(y) & \text{if } sig(x)sig(y) < 2 \\ expo(x) + expo(y) + 1 & \text{if } sig(x)sig(y) \geq 2 \end{cases}$;
(c) $sig(xy) = \begin{cases} sig(x)sig(y) & \text{if } sig(x)sig(y) < 2 \\ sig(x)sig(y)/2 & \text{if } sig(x)sig(y) \geq 2. \end{cases}$

Proof

(a) $sgn(xy) = xy/|xy| = (x/|x|)(y/|y|) = sgn(x)sgn(y)$.
(b) Since $2^{expo(x)} \leq |x| < 2^{expo(x)+1}$ and $2^{expo(y)} \leq |y| < 2^{expo(y)+1}$, we have

$$\begin{aligned} 2^{expo(x)+expo(y)} &= 2^{expo(x)}2^{expo(y)} \\ &\leq |xy| \\ &< 2^{expo(x)+1}2^{expo(y)+1} \\ &= 2^{expo(x)+expo(y)+2}. \end{aligned}$$

If $sig(x)sig(y) = |x|2^{-expo(x)}|y|2^{-expo(y)} < 2$, then

$$|xy| < 2 \cdot 2^{expo(x)}2^{expo(y)} = 2^{expo(x)+expo(y)+1},$$

and by Definition 4.1, $expo(xy) = expo(x) + expo(y)$.
On the other hand, if $sig(x)sig(y) \geq 2$, then similarly,

$$|xy| \geq 2^{expo(x)+expo(y)+1},$$

and Definition 4.1 yields $expo(xy) = expo(x) + expo(y) + 1$.
(c) If $sig(x)sig(y) < 2$, then

$$\begin{aligned} sig(xy) &= |xy|2^{-expo(xy)} \\ &= |xy|2^{-(expo(x)+expo(y))} \\ &= |x|2^{-expo(x)}|y|2^{-expo(y)} \\ &= sig(x)sig(y). \end{aligned}$$

Otherwise,

$$\begin{aligned} sig(xy) &= |xy|2^{-expo(xy)} \\ &= |xy|2^{-(expo(x)+expo(y)+1)} \\ &= |x|2^{-expo(x)}|y|2^{-expo(y)}/2 \\ &= sig(x)sig(y)/2. \end{aligned}$$

□

4.2 Exactness

In order for a significand, which has the form

$$(1.\beta_1\beta_2\cdots)_2,$$

to be represented by an n-bit field, we must have $\beta_k = 0$ for all $k \geq n$, or equivalently, a right shift of the radix point by $n-1$ places must result in an integer. This motivates our definition of *exactness*.

Definition 4.2 If $x \in \mathbb{R}$ and $n \in \mathbb{N}$, then x is n-exact $\Leftrightarrow$ $sig(x)2^{n-1} \in \mathbb{Z}$.

The definition may be restated in various ways:

Lemma 4.8 *Let $x \in \mathbb{R}$, $n \in \mathbb{N}$, and $k \in \mathbb{Z}$. The following are equivalent:*

(a) x *is* n*-exact;*
(b) $-x$ *is* n*-exact;*
(c) $sig(x)$ *is* n*-exact;*
(d) $2^k x$ *is* n*-exact;*
(e) $x2^{n-1-expo(x)} \in \mathbb{Z}$.

It is clear that if $sig(x)$ is representable in a given field of bits, then it is also representable in any wider field.

Lemma 4.9 *For all $x \in \mathbb{R}$, $n \in \mathbb{Z}$, and $m \in \mathbb{Z}$, if $m < n$ and x is m-exact, then x is n-exact.*

Proof Since $2^{n-m} \in \mathbb{Z}$ and $x \cdot 2^{m-1-expo(x)} \in \mathbb{Z}$,

$$x \cdot 2^{n-1-expo(x)} = 2^{n-m} \cdot x \cdot 2^{m-1-expo(x)} \in \mathbb{Z}.$$

□

A power of 2 has a 1-bit significand.

Lemma 4.10 *If $n \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$, then 2^n is m-exact.*

Proof By Lemma 4.2 (d),

$$2^n \cdot 2^{m-1-expo(2^n)} = 2^n \cdot 2^{m-1-n} = 2^{m-1} \in \mathbb{Z}.$$

□

A bit vector is exact with respect to its width.

Lemma 4.11 *Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}$. If x is a bit vector of width n, then x is n-exact.*

Proof Since 0 is n-exact for all $n \in \mathbb{Z}$, we may assume $x > 0$, and therefore $n > 0$. Now since $0 < x < 2^n$, Lemma 4.2 implies $expo(x) \leq n-1$, and hence $x \cdot 2^{n-1-expo(x)} \in \mathbb{Z}$. □

We have the following formula for the exactness of a product.

Lemma 4.12 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, $m \in \mathbb{Z}$, and $n \in \mathbb{Z}$. If x is m-exact and y is n-exact, then xy is $(m+n)$-exact.*

Proof If $x2^{m-1-expo(x)}$ and $y2^{n-1-expo(y)}$ are integers, then by Lemma 4.7, so is

$$x2^{m-1-expo(x)}y2^{n-1-expo(y)}2^{expo(x)+expo(y)+1-expo(xy)} = xy2^{m+n-1-expo(xy)}.$$

□

In particular, if x is n-exact, then x^2 is $2n$-exact. The converse also holds.

Lemma 4.13 *Let $x \in \mathbb{Q}$ and $n \in \mathbb{Z}$. If x^2 is $2n$-exact, then x is n-exact.*

Proof We shall require an elementary fact of arithmetic:

$$\text{If } r \in \mathbb{Q} \text{ and } 2r^2 \in \mathbb{Z}, \text{ then } r \in \mathbb{Z}.$$

In order to establish this result, let $r = a/b$, where a and b are integers with no common prime factor, and assume that $2r^2 = 2a^2/b^2 \in \mathbb{Z}$. If b were even, say $b = 2c$, then $2a^2/b^2 = a^2/2c^2 \in \mathbb{Z}$, implying that a is even, which is impossible. Thus, b cannot have a prime factor that does not also divide a, and hence $b = 1$.

Now to show that x is n-exact, i.e., $x2^{n-1-expo(x)} \in \mathbb{Z}$, it will suffice to show that

$$2\left(x2^{n-1-expo(x)}\right)^2 = x^2 2^{2n-1-2expo(x)} \in \mathbb{Z}.$$

But since x^2 is $2n$-exact, we have $x^2 2^{2n-1-expo(x^2)} \in \mathbb{Z}$, and since Lemma 4.7 implies $expo(x^2) \geq 2expo(x)$,

$$x^2 2^{2n-1-2expo(x)} = x^2 2^{2n-1-expo(x^2)} 2^{expo(x^2)-2expo(x)} \in \mathbb{Z}.$$

□

Lemma 4.14 *Let $n \in \mathbb{N}$, $x \in \mathbb{R}$, and $y \in \mathbb{R}$. Assume that x and y are nonzero and k-exact for some $k \in \mathbb{N}$. If xy is n-exact, then so are x and y.*

Proof Let p and q be the smallest integers such that $x' = 2^p x$ and $y' = 2^q y$ are integers. Thus, x' and y' are odd. By Lemma 4.8, x, y, or xy, respectively, is n-exact iff x', y', or $x'y'$ is n-exact. Consequently, we may replace x and y with x' and y'. That is, we may assume without loss of generality that x and y are odd integers.

By Lemma 4.8, an odd integer z is n-exact iff $expo(z) \leq n-1$, or equivalently, according to Lemma 4.2, $|z| < 2^n$. Thus, since xy is n-exact, $xy < 2^n$, which implies $x < 2^n$ and $y < 2^n$, and hence x and y are n-exact. □

Exactness of a bit vector may be formulated in various ways.

Lemma 4.15 *Let $x \in \mathbb{N}$, $k \in \mathbb{N}$, and $n \in \mathbb{N}$. If $expo(x) = n-1$ and $k < n$, then the following are equivalent:*

(a) x *is* $(n-k)$*-exact;*
(b) $x/2^k \in \mathbb{Z}$*;*
(c) $x[n-1:k] = x/2^k$*;*
(d) $x[k-1:0] = 0$.

Proof The equivalence between (a) and (b) follows from Lemma 4.8, since

$$x2^{(n-k)-1-expo(x)} = x2^{n-k-1-(n-1)} = x/2^k.$$

Now by Lemmas 2.4 and 2.18,

$$x = x[n-1:0] = 2^k x[n-1:k] + x[k-1:0],$$

and hence

$$x/2^k = x[n-1:k] + x[k-1:0]/2^k.$$

But by Lemma 2.1, $0 \leq x[k-1:0]/2^k < 1$, which implies that (b), (c), and (d) are all equivalent. □

Corollary 4.16 *Let* $x \in \mathbb{N}$*,* $k \in \mathbb{N}$*, and* $n \in \mathbb{N}$ *such that* $expo(x) = n-1$ *and* $k < n$*, Assume that* x *is* $(n-k)$*-exact. Then* x *is* $(n-k-1)$*-exact if and only if* $x[k] = 0$.

The next lemma gives a formula for exactness of a difference.

Lemma 4.17 *Let* $x \in \mathbb{R}$*,* $y \in \mathbb{R}$*,* $n \in \mathbb{Z}$*, and* $k \in \mathbb{Z}$*. Assume that* $n > 0$ *and* $n > k$*. If* x *and* y *are both* n*-exact and* $expo(x-y) + k \leq min(expo(x), expo(y))$*, then* $x - y$ *is* $(n-k)$*-exact.*

Proof Since x is n-exact and $expo(x-y) + k \leq expo(x)$,

$$x2^{n-1-(expo(x-y)+k)} = x2^{n-1-expo(x)}2^{expo(x)-(expo(x-y)+k} \in \mathbb{Z}$$

by Lemma 4.8. Similarly, $y2^{n-1-(expo(x-y)+k)} \in \mathbb{Z}$. Thus,

$$(x-y)2^{(n-k)-1-expo(x-y)} = x2^{n-1-(expo(x-y)+k)} - y2^{n-1-(expo(x-y)+k)} \in \mathbb{Z}.$$

□

Lemma 4.17 is often applied with $k = 0$, in the following weaker form.

Corollary 4.18 *Let* $x \in \mathbb{R}$*,* $y \in \mathbb{R}$*, and* $n \in \mathbb{Z}$ *with* $n > 0$*. If* x *and* y *are both* n*-exact and* $|x-y| \leq min(|x|, |y|)$*, then* $x - y$ *is* n*-exact.*

If x is positive and n-exact, then the least n-exact number exceeding x is computed as follows.

Definition 4.3 Let $x \in \mathbb{R}$, $x > 0$, and $n \in \mathbb{N}$, $n > 0$. Then

$$fp^+(x, n) = x + 2^{expo(x)+1-n}.$$

Lemma 4.19 *Let $x \in \mathbb{R}$, $x > 0$, and $n \in \mathbb{N}$, $n > 0$. If x is n-exact, then $fp^+(x,n)$ is n-exact.*

Proof Let $e = expo(x)$ and $x^+ = fp^+(x,n) = x + 2^{e+1-n}$. Since x is n-exact, $x \cdot 2^{n-1-e} \in \mathbb{Z}$, and by Lemma 4.2, $x < 2^{e+1}$. Thus,

$$x \cdot 2^{n-1-e} < 2^{e+1} \cdot 2^{n-1-e} = 2^n,$$

which implies $x \cdot 2^{n-1-e} \leq 2^n - 1$. Therefore, $x \leq 2^{e+1} - 2^{e+1-n}$ and $x^+ = x + 2^{e+1-n} \leq 2^{e+1}$. If $x^+ = 2^{e+1}$, then x^+ is n-exact by Lemma 4.10. Otherwise, $x^+ < 2^{e+1}$, $expo(x^+) = e$, and

$$x^+ \cdot 2^{n-1-expo(x^+)} = (x + 2^{e+1-n})2^{n-1-e} = x \cdot 2^{e+1-n} + 1 \in \mathbb{Z}.$$

□

Lemma 4.20 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, $y > x > 0$, and $n \in \mathbb{N}$, $n > 0$. If x and y are n-exact, then $y \geq fp^+(x,n)$.*

Proof Let $e = expo(x)$ and $x^+ = fp^+(x,n) = x + 2^{e+1-n}$. Since x is n-exact, $x \cdot 2^{n-1-e} \in \mathbb{Z}$. Similar, since y is n-exact, $y \cdot 2^{n-1-expo(y)} \in \mathbb{Z}$. But $expo(y) \geq e$ by Lemma 4.2 (c), and hence

$$y \cdot 2^{n-1-e} = y \cdot 2^{n-1-expo(y)} \cdot 2^{expo(y)-e} \in \mathbb{Z}$$

as well. Now since $y > x$, $y \cdot 2^{n-1-e} > x \cdot 2^{n-1-e}$, which implies

$$y \cdot 2^{n-1-e} \geq x \cdot 2^{n-1-e} + 1.$$

Thus,

$$y \geq (x \cdot 2^{n-1-e} + 1)/2^{n-1-e} = x + 2^{e+1-n} = x^+.$$

□

Corollary 4.21 *Let $x \in \mathbb{R}$, $x > 0$, and $n \in \mathbb{N}$, $n > 0$. If x is n-exact and $expo(fp^+(x,n)) \neq expo(x)$, then $fp^+(x,n) = 2^{expo(x)+1}$.*

Proof Since $fp^+(x,n) > x$, Lemma 4.2 (c) implies $expo(fp^+(x,n)) \geq expo(x)$. Therefore, we have $expo(fp^+(x,n)) > expo(x)$, which, according to Lemma 4.2, implies $fp^+(x,n) \geq 2^{expo(x)+1}$. On the other hand, since $2^{expo(x)+1}$ is n-exact by Lemma 4.10, Lemma 4.20 implies $2^{expo(x)+1} \geq fp^+(x,n)$. □

Corollary 4.22 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{N}$ with $n > 0$. If x and y are n-exact and $x \neq y$, then*

$$expo(x - y) \geq min(expo(x), expo(y)) + 1 - n$$

Proof Since the case $xy < 0$ is trivial, we shall assume that $xy > 0$ and, without loss of generality, that $y > x > 0$. But then by Lemma 4.20,

$$expo(y - x) \geq expo(2^{expo(x)+1-n}) = expo(x) + 1 - n.$$

□

Similarly, the following definition computes the greatest n-exact number that is less than a given n-exact number.

Definition 4.4 Let $x \in \mathbb{R}$, $x > 0$, and $n \in \mathbb{N}$, $n > 0$. Then

$$fp^-(x, n) = \begin{cases} x - 2^{expo(x)-n} & \text{if } x = 2^{expo(x)} \\ x - 2^{expo(x)+1-n} & \text{if } x \neq 2^{expo(x)}. \end{cases}$$

.

Lemma 4.23 *Let $x \in \mathbb{R}$, $x > 0$, and $n \in \mathbb{N}$, $n > 0$. If x is n-exact, then $fp^-(x, n)$ is n-exact.*

Proof Let $e = expo(x)$ and $x^- = fp^-(x, n)$.

Suppose first that $x = 2^e$. Then $x^- = x - 2^{e-n}$, and since

$$2^e > x^- \geq 2^e - 2^{e-1} = 2^{e-1},$$

Definition 4.1 implies $expo(x^-) = e - 1$. Therefore,

$$x^- 2^{n-1-expo(x^-)} = x^- 2^{n-e} = (x - 2^{e-n})2^{n-e} = (2^e - 2^{e-n})2^{n-e} = 2^n - 1 \in \mathbb{Z},$$

and x^- is n-exact by Lemma 4.8.

Now suppose $x \neq 2^e$. Then by Lemma 4.19,

$$x \geq fp^+(2^e, n) = 2^e + 2^{e+1-n}.$$

Now $x^- = x - 2^{e+1-n} \geq 2^e$, and hence $expo(x^-) = e$. Thus,

$$x^- 2^{n-1-expo(x^-)} = x^- 2^{n-1-e} = (x - 2^{e+1-n})2^{n-1-e} = x2^{n-1-e} - 1 \in \mathbb{Z},$$

which implies x^- is n-exact. □

Lemma 4.24 *Let $x \in \mathbb{R}$, $x > 0$, and $n \in \mathbb{N}$, $n > 0$. If x is n-exact, then*

$$fp^+(fp^-(x, n), n) = fp^-(fp^+(x, n), n) = x.$$

Proof Let $e = expo(x)$, $x^- = fp^-(x, n)$, and $x^+ = fp^+(x, n)$. We shall prove first that $fp^+(x^-, n) = x$.

As noted in the proof of Lemma 4.19,

$$expo(x^-) = \begin{cases} e - 1 & \text{if } x = 2^e \\ e & \text{if } x > 2^e. \end{cases}$$

If $x = 2^e$, then

$$fp^+(x^-, n) = x^- + 2^{(e-1)+1-n} = x - 2^{e-n} + 2^{e-n} = x.$$

But if $x > 2^e$, then

$$fp^+(x^-, n) = x^- + 2^{e+1-n} = x - 2^{e+1-n} + 2^{e+1-n} = x.$$

Next, we prove that $fp^-(x^+, n) = x$. Note that $x^+ = x + 2^{e+1-n}$. If $x^+ < 2^{e+1}$, then $expo(x^+) = e$, and since $x^+ \neq 2^e$,

$$fp^-(x^+, n) = x^+ - 2^{e+1-n} = x + 2^{e+1-n} - 2^{e+1-n} = x.$$

We may assume, therefore, that $x^+ \geq 2^{e+1}$. By Lemma 4.19, since 2^{e+1} is n-exact, $2^{e+1} \geq x^+$, and hence, $x^+ = 2^{e+1}$. Thus,

$$fp^-(x^+, n) = x^+ - 2^{e+1-n} = x + 2^{e+1-n} - 2^{e+1-n} = x.$$

□

Lemma 4.25 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, $x > y > 0$, and $n \in \mathbb{N}$, $n > 0$. If x and y are n-exact, then $y \leq fp^-(x, n)$.*

Proof Suppose $y > fp^-(x, n)$. Then since $fp^-(x, n)$ and y are both n-exact, Lemma 4.19 implies $y \geq fp^+(fp^-(x, n), n) = x$, a contradiction. □

Chapter 5
Floating-Point Formats

A floating-point format is a scheme for representing a number as a bit vector consisting of three fields corresponding to its sign, exponent, and significand. In this chapter, we present a classification of such formats, including those prescribed by IEEE Standard 754 [9], and examine the characteristics of the numbers that they represent.

5.1 Classification of Formats

A floating-point format is characterized by the precision p with which representable numbers are differentiated, and the number q of bits allocated to the exponent, determining the range of representable numbers. Some formats represent all p bits of a number's significand explicitly, but a common optimization is to omit the integer bit. Thus, we define a format as a triple:

Definition 5.1 A floating-point format is a triple $F = \langle e, p, q \rangle$, where

(a) e is a boolean indication that the format is explicit ($e = 1$) or implicit ($e = 0$), i.e., whether or not the integer bit is explicitly represented;
(b) $p = prec(F)$ is an integer, $p \geq 2$, the precision of F;
(c) $q = expw(F)$ is an integer, $q \geq 2$, the exponent width of F.

The significand width of F is $sigw(F) = \begin{cases} p & \text{if } F \text{ is explicit} \\ p - 1 & \text{if } F \text{ is implicit.} \end{cases}$

In this chapter, every "format" will be understood to be a floating-point format.

Definition 5.2 An encoding for a format F is a bit vector of width $expw(F) + sigf(F) + 1$.

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_5

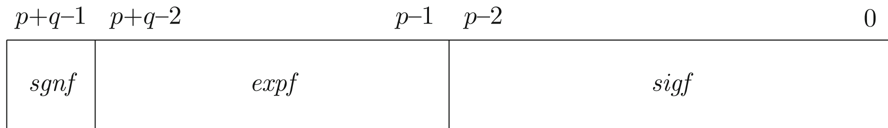


Fig. 5.1 A floating-point format with implicit integer bit

Definition 5.1 is sufficiently broad to include the formats that will be of interest to us. The most common implicit formats are the IEEE *basic single* ($p = 24$, $q = 8$) and *double precision* ($p = 53$, $q = 11$) formats, at least one of which must be implemented by any IEEE-compliant floating-point unit. Explicit formats include most implementations of the *single extended* ($p = 32$, $q = 11$) and *double extended* ($p = 64$, $q = 15$) formats, as well as the higher-precision formats that are typically used for internal computations in floating-point units.

We establish the following notation for the formats that are used by the x86 and Arm elementary arithmetic operations discussed in Part IV:

Definition 5.3 The half, single, double, and (double) extended formats are as follows:

$$HP = \langle 0, 11, 5\rangle;$$

$$SP = \langle 0, 24, 8\rangle;$$

$$DP = \langle 0, 53, 11\rangle;$$

$$EP = \langle 1, 64, 15\rangle.$$

The *sign*, *exponent*, and *significand* fields of an encoding are defined as illustrated in Figs. 5.1 and 5.2. We also define the *mantissa* field as the significand field without the integer bit, if present:

Definition 5.4 If x is an encoding for a format F, then

(a) $sgnf(x, F) = x[expw(F) + sigw(F)]$;
(b) $expf(x, F) = x[expw(F) + sigw(F)-1 : sigw(F)]$;
(c) $sigf(x, F) = x[sigw(F)-1]$;
(d) $manf(x, F) = x[prec(F) - 2]$.

The encodings for a given format are partitioned into several classes determined primarily by the exponent field, as described in the next three sections.

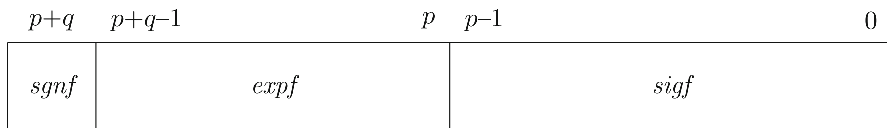


Fig. 5.2 A floating-point format with explicit integer bit

5.2 Normal Encodings

Encodings with exponent field 0 represent very small values according to a separate scheme (Sect. 5.3); those with maximal exponent field (all 1s) are reserved for representing non-numerical entities (Sect. 5.4). The remaining encodings are the subject of this section.

Definition 5.5 An encoding x for a format F is *normal* iff the following conditions hold:

(a) $0 < expf(x, F) < 2^{expw(F)} - 1$;
(b) If F is explicit, then $x[prec(F){-}1] = 1$.

The implied integer bit of a normal encoding for an implicit format is 1, as reflected in Definition 5.8 below. For an explicit format, the case of a nonzero exponent field and a zero leading significand bit is an anomaly that should never be generated by hardware.

Definition 5.6 An encoding x for an explicit format F is *unsupported* iff the following conditions hold:

(a) $expf(x, F) > 0$;
(b) $x[prec(F){-}1] = 0$.

Let x be a normal encoding for a format F with $prec(F) = p$ and $expw(F) = q$. The significand field $sigf(x, F)$ is interpreted as a p-exact value in the interval $[1, 2)$, i.e., with an implied binary point following the leading bit, which is 1, either explicitly or implicitly. The value encoded by x is the signed product of this value and a power of 2 determined by the exponent field. Since it is desirable for the range of exponents to be centered at 0, this field is interpreted with a bias of $2^{q-1} - 1$, i.e., the value of the exponent represented is

$$2^{expf(x,F)-(2^{q-1}-1)},$$

which lies in the range

$$2^{q-1} - 1 \leq expf(x, F) - (2^{q-1} - 1) \leq 2^{q-1}.$$

Definition 5.7 The exponent bias of a format F is $bias(F) = 2^{expw(F)-1} - 1$.

Thus, the decoding function is defined as follows.

Definition 5.8 Let F be a format with $p = prec(F)$ and $B = bias(F)$. If x is a normal encoding for F, then

$$ndecode(x, F) = (-1)^{sgnf(x,F)} \left(1 + 2^{1-p} manf(x, F)\right) 2^{expf(x,F)-B}.$$

The following is a trivial consequence of Definition 5.8.

Lemma 5.1 *Let x be a normal encoding for a format F and let $\hat{x} = ndecode(x, F)$. then*

(a) $sgn(\hat{x}) = (-1)^{sgnf(x,F)}$;
(b) $expo(\hat{x}) = expf(x, F) - bias(F)$;
(c) $sig(\hat{x}) = 1 + 2^{1-prec(F)} manf(x, F)$.

The numbers that admit normal encodings may be characterized as follows:

Definition 5.9 Let F be a format and let $r \in \mathbb{Q}$. Then r is a *normal value* of F iff the following conditions hold:

(a) $r \neq 0$;
(b) $0 < expo(r) + bias(F) < 2^{expw(F)} - 1$;
(c) r is $prec(F)$-exact.

The normal encoding of a normal value is derived as follows.

Definition 5.10 Let r be a normal value of F. Let

$$s = \begin{cases} 0 \text{ if } r > 0 \\ 1 \text{ if } r < 0, \end{cases}$$

$$e = expo(r) + bias(F),$$

and

$$m = 2^{prec(F)-1} sig(r).$$

then

$$nencode(r, F) = \{1'\, s, expw(F)'\, e, sigw(F)'\, m\}.$$

The next two lemmas establish an inverse relation between the encoding and decoding functions, from which it follows that the numbers that admit normal encodings are precisely those that satisfy Definition 5.9.

Lemma 5.2 *If x is a normal encoding for a format F, then $ndecode(x, F)$ is a normal value of F and*

$$nencode(ndecode(x, F), F) = x.$$

Proof Let $p = prec(F)$, $q = expw(F)$, $B = bias(F)$, and $\hat{x} = ndecode(x, F)$. It is clear from Definition 5.8 that $\hat{x} \neq 0$. By Lemma 5.1,

$$expo(\hat{x}) + B = expf(x, F)$$

is a q-bit vector, and

$$2^{p-1} sig(\hat{x}) = 2^{p-1}(1 + 2^{1-p} sigf(x, F)) = 2^{p-1} + sigf(x, F) \in \mathbb{Z},$$

i.e., $\hat{x}$ is p-exact. Thus, $\hat{x}$ is a normal value of F.

It also clear from Definition 5.8 that $sgnf(x, F) = \begin{cases} 0 \text{ if } \hat{x} > 0 \\ 1 \text{ if } \hat{x} < 0. \end{cases}$
Suppose F is implicit. Then by Definitions 5.10 and 5.4 and Lemmas 2.44 and 2.4,

$$\begin{aligned} nencode(\hat{x}, F) &= \{sgnf(x, F), q\,'\,(expo(\hat{x}) + B)\}, (p-1)\,'\,(2^{p-1}(sig(\hat{x}) - 1))\} \\ &= \{sgnf(x, F), q\,'\,expf(x, F), (p-1)\,'\,sigf(x, p)\} \\ &= \{x[p+q-1], x[p+q-2 : p-1], x[p-2 : 0]\} \\ &= x[p+q-1 : 0] \\ &= x. \end{aligned}$$

The explicit case is similar. □

Lemma 5.3 *If r is a normal value of a format F, then $nencode(r, F)$ is a normal encoding for F and*

$$ndecode(nencode(r, F), F) = r.$$

Proof We give the proof for the implicit case; the explicit case is similar.

Let $p = prec(F)$, $q = expw(F)$, $B = bias(F)$, and $x = nencode(r, F)$. By Lemma 2.42, x is a $(p + q)$-bit vector and by Lemma 2.46,

$$sgnf(x, F) = x[p+q-1] = \begin{cases} 0 \text{ if } r > 0 \\ 1 \text{ if } r < 0, \end{cases}$$

$$expf(x, F) = x[p+q-2 : p-1] = (expo(r) + B)[q-1 : 0],$$

and

$$sigf(x, F) = x[p-2 : 0] = (2^{p-1}(sig(r) - 1))[p-2 : 0].$$

Since $expo(r) + B$ is a q-bit vector,

$$(expo(r) + B)[q-1 : 0] = expo(r) + B$$

by Lemma 2.4.

Since r is p-exact,

$$2^{p-1}(sig(r) - 1) = 2^{p-1}sig(r) - 2^{p-1} \in \mathbb{Z}$$

and by Lemma 4.3, $2^{p-1}(sig(r) - 1) < 2^{p-1}$, which implies

$$(2^{p-1}(sig(r) - 1))[p - 2 : 0] = 2^{p-1}(sig(r) - 1).$$

Finally, according to Definition 5.8,

$$\begin{aligned} ndecode(x, F) &= (-1)^{sgnf(x,F)}(2^{p-1} + sigf(x, p))2^{expf(x,F)+1-p-bias(F)} \\ &= sgn(r)2^{p-1}sig(r)2^{expo(r)+B+1-p-B} \\ &= sgn(r)sig(r)2^{expo(r)} \\ &= r. \end{aligned}$$

□

We shall have occasion to refer to the smallest and largest positive numbers that admit normal representations.

Definition 5.11 The *smallest positive normal* of a format F is

$$spn(F) = 2^{1-bias(F)} = 2^{2-2^{expw(F)-1}}.$$

Lemma 5.4 *For any format F,*

(a) $spn(F) > 0$;
(b) $spn(F)$ is a normal value of F;
(c) r is a normal value of F, then $|r| \geq spn(F)$.

Proof It is clear that $spn(F)$ is positive and satisfies Definition 5.9. Moreover, if $r > 0$ and r is a normal value of F, then since $expo(r) > -bias(F)$, $r \geq 2^{1-bias(F)}$ by Lemma 4.2. □

Definition 5.12 The *largest positive normal* of a format F is

$$lpn(F) = 2^{2^{expw(F)}-2-bias(F)}(2 - 2^{1-prec(F)}).$$

Lemma 5.5 *For any format F,*

(a) $lpn(F) > 0$;
(b) $lpn(F)$ is a normal value of F;
(c) If r is a normal value of F, then $r \leq lpn(F)$.

Proof It is clear that $lpn(F)$ is positive and satisfies Definition 5.9. Let $p = prec(F)$ and $q = expw(F)$. If r is a normal value of F, then by Definition 5.9, $expo(r) \leq 2^q - 2 - bias(F)$ and r is p-exact. Thus, $r < 2^{expo(r)+1} \leq 2^{2^q-1-bias(F)}$ and by Lemma 4.25,

$$r \leq fp^-(2^{2^q-1-bias(q)}, p) = lpn(F).$$

□

5.3 Denormals and Zeroes

An exponent field of 0 is used to encode numerical values that lie below the normal range. If the exponent and significand fields of an encoding are both 0, then the encoded value itself is 0 and the encoding is said to be a *zero*. If the exponent field is 0 and the significand field is not, then the encoding is either *denormal* or *pseudo-denormal*:

Definition 5.13 Let x be an encoding for a format F with $expf(x, F) = 0$.

(a) If $sigf(x, F) = 0$, then x is a *zero encoding for F*.
(b) If $sigf(x, F) \neq 0$ and either F is implicit or $x[prec(F)-1] = 0$, then x is a *denormal encoding for F*.
(c) If F is explicit and $x[prec(F)-1] = 1$, then x is a *pseudo-denormal* encoding for F.

Note that a zero can have either sign:

Definition 5.14 Let F be a format and let $s \in \{0, 1\}$. Then

$$zencode(s, F) = \{1'\,s, (expw(F) + sigw(F))'\,0\}.$$

The numerical value of a denormal encoding is interpreted according to a separate scheme that provides a broader range (at the expense of lower precision) than would otherwise be possible. There are two differences between the decoding formulas for denormal and normal encodings:

1. For a denormal encoding for an implicit format, the integer bit is taken to be 0 rather than 1.
2. The power of 2 represented by the zero exponent field of a denormal or pseudo-denormal encoding is $2^{1-bias(F)}$ rather than $2^{0-bias(F)}$.

Definition 5.15 Let F be a format with $p = prec(F)$ and $B = bias(F)$. If x is an encoding for F with $expf(x, F) = 0$, then

$$\begin{aligned} ddecode(x, F) &= (-1)^{sgnf(x,F)} \left(2^{1-p} sigf(x, F)\right) 2^{1-B} \\ &= (-1)^{sgnf(x,F)} sigf(x, F) 2^{2-p-B}. \end{aligned}$$

We also define a general decoding function:

Definition 5.16 Let x be an encoding for a format F. If $\mathit{expf}(x, F) \neq 2^{\mathit{expw}(F)} - 1$, then x is a *numerical* encoding and

$$\mathit{decode}(x, F) = \begin{cases} \mathit{ndecode}(x, F) \text{ if } \mathit{expf}(x, F) \neq 0 \\ \mathit{ddecode}(x, F) \text{ if } \mathit{expf}(x, F) = 0. \end{cases}$$

Note that the function *ddecode* is applied to pseudo-denormal as well as denormal encodings. If x is a pseudo-denormal encoding for an explicit format F and x' is the normal encoding derived from x by replacing its 0 exponent field with 1, then expanding Definitions 5.8 and 5.15 and observing that $\mathit{sigf}(x, F) = 2^{p-1} + \mathit{manf}(x, F)$, we have

$$\mathit{decode}(x', F) = \mathit{ndecode}(x', F) = \mathit{ddecode}(x, F) = \mathit{decode}(x, F).$$

Thus, any value encoded as a pseudo-denormal admits an alternative encoding as a normal.

Lemma 5.6 *Let x be a denormal encoding for F and let $\hat{x} = \mathit{ddecode}(x, F)$.*

(a) $\mathit{sgn}(\hat{x}) = (-1)^{\mathit{sgnf}(x,F)}$.
(b) $\mathit{expo}(\hat{x}) = \mathit{expo}(\mathit{sigf}(x, F)) - \mathit{bias}(F) + 2 - \mathit{prec}(F)$.
(c) $\mathit{sig}(\hat{x}) = \mathit{sig}(\mathit{sigf}(x, F))$.

Proof (a) is trivial; (b) and (c) follow from Lemmas 4.5 and 4.6. □

The class of numbers that are representable as denormal encodings is recognized by the following predicate.

Definition 5.17 Let F be a format and let $r \in \mathbb{Q}$. Then r is a *denormal value* of F iff the following conditions hold:

(a) $r \neq 0$;
(b) $2 - \mathit{prec}(F) \leq \mathit{expo}(r) + \mathit{bias}(F) \leq 0$;
(c) r is $(\mathit{prec}(F) + \mathit{expo}(r) - \mathit{expo}(\mathit{spn}(F)))$-exact.

The encoding of a denormal value is constructed as follows.

Definition 5.18 If r is a denormal value of F with

$$s = \begin{cases} 0 \text{ if } r > 0 \\ 1 \text{ if } r < 0 \end{cases}$$

and

$$m = 2^{\mathit{prec}(F)-2+\mathit{expo}(r)+\mathit{bias}(F)} \mathit{sig}(r),$$

then

$$\mathit{dencode}(r, F) = \{1\,'\,s, \mathit{expw}(F)\,'\,0, \mathit{sigw}(F)\,'\,m\}.$$

According to the next two lemmas, *ddecode* and *dencode* are inverse functions and the denormal values are precisely those that are represented by denormal encodings.

Lemma 5.7 *If x is a denormal encoding for a format F, then $ddecode(x, F)$ is a denormal value of F and*

$$dencode(ddecode(x, F), F) = x.$$

Proof Let $p = prec(F)$, $q = expw(F)$, $b = bias(F)$, $s = sgnf(x, F)$, $m = sigf(x, F)$, and $\hat{x} = ddecode(x, F)$. Since $1 \leq m < 2^{p-1}$,

$$2^{2-p-b} \leq |\hat{x}| = 2^{2-p-b} m < 2^{1-b},$$

and by Lemma 4.2,

$$2 - p - b \leq expo(\hat{x}) < 1 - b,$$

which is equivalent to Definition 5.17 (b). In order to prove (c), we must show, according to Definition 4.2, that

$$2^{p+expo(r)-expo(spn(F)+expo(\hat{x})-1} sig(\hat{x}) = 2^{p-2+b+expo(\hat{x})} sig(\hat{x}) \in \mathbb{Z}.$$

But

$$\begin{aligned} 2^{p-2+b+expo(\hat{x})} sig(\hat{x}) &= 2^{p-2+b+expo(\hat{x})} |\hat{x}| 2^{-expo(\hat{x})} \\ &= 2^{p-2+b} |\hat{x}| \\ &= 2^{p-2+b} (2^{2-p-b} m) \\ &= m \in \mathbb{Z}. \end{aligned}$$

This establishes that $\hat{x}$ is a denormal value.

Now by Definition 5.15, $s = \begin{cases} 0 \text{ if } \hat{x} > 0 \\ 1 \text{ if } \hat{x} < 0. \end{cases}$

Therefore, by Definitions 5.13, 5.18, and 5.4 and Lemmas 2.44 and 2.4,

$$\begin{aligned} dencode(\hat{x}, F) &= \{1'\, s, q'\, 0, (2^{p-2+expo(\hat{x})+b} sig(\hat{x}))[p-2:0]\} \\ &= \{1'\, sgnf(x, F), expw(F)'\, expf(x, F), sigw(F)'\, sigf(x, F)\} \\ &= x. \end{aligned}$$

□

Lemma 5.8 *If r is a denormal value of F, then $dencode(r, F)$ is a denormal encoding for F and*

$$ddecode(dencode(r, F), F) = r.$$

Proof Let $p = prec(F)$, $q = expw(F)$, $b = bias(F)$, and $x = dencode(r, F)$. By Lemma 2.42, x is a $(p+q)$-bit vector and by Lemma 2.46,

$$sgnf(x, F) = x[p+q-1] = \begin{cases} 0 \text{ if } r > 0 \\ 1 \text{ if } r < 0, \end{cases}$$

$$expf(x, F) = x[p+q-2 : p-1] = 0,$$

and

$$sigf(x, F) = x[p-2:0] = (2^{p-2+expo(r)+bias(F)} sig(r))[p-2:0].$$

Since r is $(p-2+2^{q-1}+expo(r))$-exact,

$$2^{p-2+expo(r)+b} sig(r) = 2^{(p-2+2^{q-1}+expo(r))-1} sig(r) =\in \mathbb{Z}$$

and since $expo(r) + b \leq 0$,

$$2^{p-2+expo(r)+b} sig(r) < 2^{p-2} \cdot 2 = 2^{p-1}$$

by Lemma 4.3, which implies

$$(2^{p-2+expo(r)+b} sig(r))[p-2:0] = 2^{p-2+expo(r)+b} sig(r).$$

Finally, according to Definition 5.15,

$$\begin{aligned} ddecode(x, F) &= (-1)^{sgnf(x,F)} sigf(x, F) 2^{2-p-b} \\ &= sgn(r) 2^{p-2+expo(r)+b} sig(r) 2^{2-p-b} \\ &= sgn(r) sig(r) 2^{expo(r)} \\ &= r. \end{aligned}$$

□

Definition 5.19 The smallest positive denormal of a format F is

$$spd(F) = 2^{2-bias(F)-prec(F)} = 2^{3-2^{expw(F)}-prec(F)}.$$

Lemma 5.9 *For any format F,*

(a) $spd(F) > 0$;
(b) $spd(F)$ is a denormal value of F;
(c) If r is a denormal value of F, then $|r| \geq spd(F)$.

Proof Let $p = prec(F)$, $q = expw(F)$, and $b = bias(F)$. It is clear that $spd(F)$ is positive. To show that $spd(F)$ is $(p + expo(spd(F)) - expo(spn(F)))$-exact, we need only observe that

$$p + expo(spd(F)) - expo(spn(F)) = p + (2 - b - p) - (1 - b) = 1.$$

Finally, since

$$expo(spd(F)) + b = 2 - p < 0,$$

$spd(F)$ is a denormal value and moreover, $spd(F)$ is the smallest positive r that satisfies $2 - p \leq expo(r) + b$. □

Every number with a denormal representation is a multiple of the smallest positive denormal.

Lemma 5.10 *If $r \in \mathbb{Q}$ and let F be a format, then r is a denormal value of F iff $r = m \cdot spd(F)$ for some $m \in \mathbb{N}$, $1 \leq m < 2^{prec(F)-1}$.*

Proof Let $p = prec(F)$ and $b = bias(F)$. For $1 \leq m \leq p-1$, let $a_m = m \cdot spd(F)$. Then $a_1 = spd(F)$ and

$$a_{2^{p-1}} = 2^{p-1}spd(F) = 2^{p-1}2^{2-b-p} = 2^{1-b} = spn(F).$$

We shall show, by induction on m, that a_m is a denormal value of F for $1 \leq m < 2^{n-1}$. First note that for all such m,

$$\begin{aligned}
&fp^+(a_m, p + expo(a_m) - expo(spn(F))) \\
&= a_m + 2^{expo(a_m)+1-(p+expo(a_m)-expo(spn(F)))} \\
&= a_m + 2^{expo(spn(F))-(p-1)} \\
&= a_m + spd(F) \\
&= a_{m+1}.
\end{aligned}$$

Suppose that a_{m-1} is a denormal value for some m, $1 < m < 2^{p-1}$. Then a_{m-1} is $(p + expo(a_{m-1}) - expo(spn(F)))$-exact, and by Lemma 4.20, so is a_m. But since $expo(a_m) \geq expo(a_{m-1})$, it follows from Lemma 4.9 that a_m is also $(p+expo(a_m)-expo(spn(F)))$-exact. Since

$$a_m < a_{2^{p-1}} = spn(F) = 2^{1-b},$$

$expo(a_m) < 1 - b$, i.e., $expo(a_m) + b \leq 0$, and hence, a_m is a denormal value.

Now suppose that z is a denormal value. Let $m = \lfloor z/a_1 \rfloor$. Clearly, $1 \leq m < 2^{p-1}$, and $a_m \leq z < a_{m+1}$. It follows from Lemma 4.21 that $expo(z) = expo(a_m)$, and consequently, z is $(p+expo(a_m)-expo(spn(F)))$-exact. Thus, by Lemma 4.20, $z = a_m$. □

5.4 Infinities and NaNs

The upper extreme value $2^{expw(F)} - 1$ of the exponent field is reserved for encoding non-numerical entities. According to Definition 5.6, an encoding for an explicit format with exponent field $2^{expw(F)} - 1$ is unsupported if its integer bit is 0. In all other cases an encoding with this exponent field is an *infinity* if its significand field is 0, and a *NaN* ("Not a Number") otherwise. A NaN is further classified as an *SNaN* ("signaling NaN") or a *QNaN* ("quiet NaN") according to the most significant bit of its mantissa field:

Definition 5.20 Let x be an encoding for a format F with $expf(x, F) = 2^{expw}(F) - 1$ and assume that if F is explicit, then $x[prec(F) - 1] = 1$.

(a) x is an infinity for F iff $expf(x, F) = manf(x, F) = 0$;
(b) x is a NaN for F iff $manf(x, F) \neq 0$;
(c) x is an SNaN for F iff x is a NaN and $x[prec(F) - 2] = 0$;
(d) x is a QNaN for F iff x is a NaN and $x[prec(F) - 2] = 1$.

An infinity is used to represent a computed value that lies above the normal range; when it occurs as an operand of an arithmetic operation, it is treated as $\pm\infty$. This function constructs an infinity with a given sign:

Definition 5.21 Let F be a format, let $s \in \{0, 1\}$ and let $e = 2^{expw(F)} - 1$. Then

$$iencode(s, F) = \begin{cases} \{1\,'\,s, expw(F)\,'\,e, sigw(F)\,'\,0\} & \text{if } F \text{ is implicit} \\ \{1\,'\,s, expw(F)\,'\,e, 1\,'\,1, (sigw(F) - 1)\,'\,0\} & \text{if } F \text{ is explicit.} \end{cases}$$

An SNaN operand always triggers an exception and is converted to a QNaN to be returned as an instruction value. SNaNs are not generated by hardware, but may be written by software to indicate exceptional conditions. For example, a block of memory may be filled with SNaNs to guard against its access until it is initialized.

A QNaN may be generated by hardware to be returned by an instruction either by "quieting" an SNaN operand or as an indication of an invalid operation, such as an indeterminate form. A QNaN operand is generally propagated as the value of an instruction without signaling an exception.

The following function converts an SNaN to a QNaN:

Definition 5.22 If x is a NaN encoding for a format F, then

$$qnanize(x, F) = x \mid 2^{prec(F)-2}.$$

The following encoding, known as the *real indefinite QNaN*, is the default value used to signal an invalid operation:

Definition 5.23 Let F be a format and let $e = 2^{expw(F)} - 1$. Then

$$indef(F) = \begin{cases} \{1\,'\,0, expw(F)\,'\,e, 1\,'\,1, (sigw(F) - 1)\,'\,0\} \text{ if } F \text{ is implicit} \\ \{1\,'\,0, expw(F)\,'\,e, 2\,'\,3, (sigw(F) - 2)\,'\,0\} \text{ if } F \text{ is explicit.} \end{cases}$$

Infinities and NaNs will be discussed further in the context of elementary arithmetic operations in Part IV.

Chapter 6
Rounding

The objective of floating-point rounding is the approximation of a real number by one that is representable in a given floating-point format. Thus, a *rounding mode* is a function that computes an n-exact value $\mathcal{R}(x, n)$, given a real number x and precision n. In this chapter, we investigate the properties of a variety of rounding modes, including those that are prescribed by the IEEE standard as well as others that are commonly used in implementations of floating-point operations.

Every mode that we consider will be shown to satisfy the following axioms, for all $x \in \mathbb{R}$, $y \in \mathbb{R}$, $n \in \mathbb{Z}^+$, and $k \in \mathbb{Z}$:

(1) $\mathcal{R}(x, n)$ is n-exact.
(2) If x is n-exact, then $\mathcal{R}(x, n) = x$.
(3) If $x \leq y$, then $\mathcal{R}(x, n) \leq \mathcal{R}(y, n)$.
(4) $\mathcal{R}(2^k x, n) = 2^k \mathcal{R}(y, n)$.

A critical consequence of these properties is that $\mathcal{R}$ is optimal in the sense that there can exist no n-exact number in the open interval between x and $\mathcal{R}(x, n)$. For example, if y is n-exact and $x < y$, then by (2) and (3), $\mathcal{R}(x, n) \leq \mathcal{R}(y, n) = y$. In particular, since 0 is n-exact for all n, it follows that

$$sgn(\mathcal{R}(x, n)) = sgn(x).$$

Also note that (4) implies that $\mathcal{R}$ is determined by its behavior for $1 \leq |x| < 2$.

In the first two sections of this chapter, we examine the basic *directed* rounding modes *RTZ* ("round toward zero") and *RAZ* ("round away from zero"), characterized by the inequalities

$$|RTZ(x, n)| \leq |x|$$

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_6

and

$$|RAZ(x, n)| \geq |x|.$$

It is clear that for any rounding mode $\mathcal{R}$ and arguments x and n, either $\mathcal{R}(x, n) = RTZ(x, n)$ or $\mathcal{R}(x, n) = RAZ(x, n)$. It is natural, therefore, to define other rounding modes in terms of these two. In Sect. 6.3, we discuss two versions of "rounding to nearest", *RNE* ("round to nearest even") and *RNA* ("round to nearest away from zero"), both of which select the more accurate of the two approximations, but which handle the ambiguous case of a midpoint between consecutive representable numbers differently.

A desirable property of a rounding mode is that the expected (i.e., average) error that it incurs is 0, so that errors generated over a long sequence of computations tend to cancel one other. In Sect. 6.3, we shall make this notion precise with the definition of an *unbiased* rounding mode, and demonstrate that it is satisfied by *RNE*, which, for this reason, is identified by the standard as the "default" rounding mode.

Among other properties that are shared by some, but not all, of the rounding modes of interest are *symmetry*,

$$\mathcal{R}(-x, n) = -\mathcal{R}(x, n),$$

and *decomposability*: the property that for $m < n$,

$$\mathcal{R}(\mathcal{R}(x, n), m) = \mathcal{R}(x, m).$$

We shall see, for example, that both versions of rounding to nearest fail to satisfy the latter condition. This accounts for the phenomenon of "double rounding": when the result of a computation undergoes a preliminary rounding to be temporarily stored in a register that is wider than the target format of an instruction, care must be taken to ensure the accuracy of the final rounding. In Sect. 6.4, we discuss another unbiased rounding mode that is commonly used internally by floating-point units to address this problem.

In Sect. 6.5, we define the two other directed rounding modes that are prescribed by the IEEE standard, and collect the properties that are shared by all IEEE modes. We present several techniques that are commonly employed in the implementation of rounding by commercial floating-point units.

Considerations other than n-exactness are involved in the rounding of results that lie outside the normal range of a format. In the case of *overflow*, which occurs when the result of a computation exceeds the representable range, the standard prescribes rounding either to the maximum representable number or to infinity. The rules that govern this choice, which are quite arbitrary from a mathematical perspective, are deferred to Part IV. The more interesting case of *underflow*, involving a denormal result, is the subject of Sect. 6.6.

In our discussion of denormal rounding (see Definition 6.10), we shall find it convenient to extend the notion of rounding to allow negative precisions. Thus, in

general, we shall consider a *rounding mode* to be a mapping

$$\mathcal{R} : \mathbb{R} \times \mathbb{Z} \to \mathbb{R}.$$

Most of the results of this chapter pertaining to $\mathcal{R}(x, n)$ will be formulated with the hypothesis $n > 0$, with the exception of those that are required in the more general case in Sect. 6.6 (see Lemmas 6.87 and 6.97).

6.1 Rounding Toward Zero

The most basic rounding mode, which the IEEE standard calls "round toward 0", may be described as a three-step operation on the significand of its first argument. Thus, $RTZ(x, n)$ has the same sign and exponent as x, while its significand is computed from $sig(x)$ as follows:

1. Shift the binary point of $sig(x)$ by $n - 1$ bits to the right.
2. Extract the floor of the result.
3. Shift the binary point by $n - 1$ bits to the left.

In other words, $RTZ(x, n)$ is the result of replacing $sig(x)$ by $\lfloor 2^{n-1} sig(x) \rfloor 2^{1-n}$:

Definition 6.1 For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}$,

$$RTZ(x, n) = sgn(x) \lfloor 2^{n-1} sig(x) \rfloor 2^{expo(x)-n+1}.$$

RTZ is also known as *truncation*; it is trivially related to the truncation function of Sect. 1.5 by

$$RTZ(x, n) = sgn(x) 2^{expo(x)} sig(x)^{(n-1)}.$$

Example Let $x = 45/8 = (101.101)_2$ and $n = 5$. Then $sgn(x) = 1$, $expo(x) = 2$, $sig(x) = (1.01101)_2$,

$$\lfloor 2^{n-1} sig(x) \rfloor 2^{1-n} = \lfloor (10110.1)_2 \rfloor 2^{-4} = 10110 \cdot 2^{-4} = 1.011,$$

and

$$RTZ(x, n) = \lfloor 2^{n-1} sig(x) \rfloor 2^{1-n} 2^{expo(x)} = 1.011 \cdot 2^2 = 101.1.$$

Note that this value is the largest 5-exact number that does not exceed x.

The second argument of any rounding mode is normally positive; for this mode, the negative-precision case is trivial:

Lemma 6.1 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}$, if $n \leq 0$, then*

$$RTZ(x, n) = 0.$$

Proof By Lemma 4.3,

$$0 < 2^{n-1} sig(x) \leq sig(x)/2 < 1,$$

which implies $\lfloor 2^{n-1} sig(x) \rfloor = 0$. □

As we have noted, *RTZ* preserves exponent:

Lemma 6.2 *If $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$, then*

$$expo(RTZ(x, n)) = expo(x).$$

Proof Since $|RTZ(x, n)| \leq |x|$, Lemma 4.2 (c) implies $expo(RTZ(x, n)) \leq expo(x)$. But by Lemmas 4.3 and 1.1,

$$|RTZ(x, n)| = \lfloor 2^{n-1} sig(x) \rfloor 2^{expo(x)-n+1} \geq 2^{n-1} 2^{expo(x)-n+1} = 2^{expo(x)},$$

and hence, by Lemma 4.2,

$$expo(RTZ(x, n)) \geq expo(x).$$

□

The following inequality is the defining characteristic of *RTZ*:

Lemma 6.3 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$,*

$$|RTZ(x, n)| \leq |x|.$$

Proof By Definition 1.1 and Lemma 4.1,

$$|RTZ(x, n)| \leq 2^{n-1} sig(x) 2^{expo(x)-n+1} = sig(x) 2^{expo(x)} = |x|.$$

□

The following complements Lemma 6.3, confining $RTZ(x, n)$ to an interval.

Lemma 6.4 *If $x \in \mathbb{R}$, $x \neq 0$, and $n \in \mathbb{Z}^+$, then*

$$|RTZ(x, n)| > |x| - 2^{expo(x)-n+1} \geq |x|(1 - 2^{1-n}).$$

Proof By Definitions 6.1 and 1.1 and Lemma 4.1,

$$|RTZ(x, n)| > (2^{n-1} sig(x) - 1) 2^{expo(x)-n+1} = |x| - 2^{expo(x)-n+1}.$$

The second inequality follows from Definition 4.1. □

Corollary 6.5 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$,*

$$|x - RTZ(x,n)| < 2^{expo(x)-n+1} \leq 2^{1-n}|x|.$$

Proof By Lemma 6.3,

$$|x - RTZ(x,n)| = ||x| - |RTZ(x,n)|| = |x| - |RTZ(x,n)|.$$

The corollary now follows from Lemma 6.4. □

The next five lemmas establish that *RTZ* has all the properties listed at the beginning of this chapter, including symmetry and decomposability:

Lemma 6.6 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}$,*

$$RTZ(-x,n) = -RTZ(x,n).$$

Proof This is an immediate consequence of Definition 6.1 □

Lemma 6.7 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$, $RTZ(x,n)$ is n-exact.*

Proof Since $expo(RTZ(x,n)) = expo(x)$, it suffices to observe that

$$RTZ(x,n)2^{n-1-expo(x)} = sgn(x)\lfloor 2^{n-1}sig(x)\rfloor \in \mathbb{Z}.$$

□

Lemma 6.8 *Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$. If x is n-exact, then*

$$RTZ(x,n) = x.$$

Proof By Definition 4.2 and Lemma 1.1,

$$\lfloor 2^{n-1}sig(x)\rfloor = 2^{n-1}sig(x),$$

and hence by Definition 6.1 and Lemma 4.1,

$$\begin{aligned}
RTZ(x,n) &= sgn(x)\lfloor 2^{n-1}sig(x)\rfloor 2^{expo(x)-n+1} \\
&= sgn(x)2^{n-1}sig(x)2^{expo(x)-n+1} \\
&= sgn(x)sig(x)2^{expo(x)} \\
&= x.
\end{aligned}$$

□

Lemma 6.9 *Let $x \in \mathbb{R}$, $a \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If a is n-exact and $a \leq x$, then $a \leq RTZ(x,n)$.*

Proof If $x < 0$, then $x \leq RTZ(x,n)$ by Lemma 6.3. Therefore, we may assume that $x \geq 0$. Suppose $a > RTZ(x,n)$. Then by Lemma 4.20,

$$RTZ(x,n) \leq a - 2^{expo(RTZ(x,n))-n+1} = a - 2^{expo(x)-n+1} \leq x - 2^{expo(x)-n+1},$$

contradicting Lemma 6.4. □

Lemma 6.10 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If $x \leq y$, then*

$$RTZ(x,n) \leq RTZ(y,n).$$

Proof First suppose $x > 0$. By Lemma 6.3, $RTZ(x,n) \leq x \leq y$. Since $RTZ(x,n)$ is n-exact by Lemma 6.7, Lemma 6.9 implies

$$RTZ(x,n) \leq RTZ(y,n).$$

Now suppose $x \leq 0$. We may assume that $x \leq y < 0$. Thus, since $0 < -y \leq -x$, we have $RTZ(-y,n) \leq RTZ(-x,n)$ and by Lemma 6.6,

$$RTZ(x,n) = -RTZ(-x,n) \leq -RTZ(-y,n) = RTZ(y,n).$$

□

Lemma 6.11 *For all $x \in \mathbb{R}$, $n \in \mathbb{Z}^+$, and $k \in \mathbb{Z}$,*

$$RTZ(2^k x, n) = 2^k RTZ(x,n).$$

Proof By Lemma 4.6,

$$\begin{aligned} RTZ(2^k x, n) &= sgn(2^k x)\lfloor 2^{n-1} sig(2^k x)\rfloor 2^{expo(2^k x)-n+1} \\ &= sgn(x)\lfloor 2^{n-1} sig(x)\rfloor 2^{expo(x)+k-n+1} \\ &= 2^k RTZ(x,n). \end{aligned}$$

□

Lemma 6.12 *Let $x \in \mathbb{R}$, $m \in \mathbb{Z}^+$, and $n \in \mathbb{Z}^+$. If $m \leq n$, then*

$$RTZ(RTZ(x,n),m) = RTZ(x,m).$$

Proof We assume $x \geq 0$; the case $x < 0$ follows easily from Lemma 6.6. Applying Lemma 1.2, we have

$$\begin{aligned} &RTZ(RTZ(x,n),m) \\ &= \lfloor 2^{m-1-expo(x)}(\lfloor 2^{n-1-expo(x)} x\rfloor 2^{expo(x)+1-n})\rfloor 2^{expo(x)+1-m} \end{aligned}$$

$$= \lfloor \lfloor 2^{n-1-expo(x)} x \rfloor / 2^{n-m} \rfloor 2^{expo(x)+1-m}$$
$$= \lfloor 2^{n-1-expo(x)} x / 2^{n-m} \rfloor 2^{expo(x)+1-m}$$
$$= \lfloor 2^{m-1-expo(x)} x \rfloor 2^{expo(x)+1-m}$$
$$= RTZ(x, m).$$

□

If x is $(n+1)$-exact but not n-exact, then x is equidistant from two successive n-exact numbers. In this case, we have an explicit formula for $RTZ(x, n)$.

Lemma 6.13 *Let $x \in \mathbb{R}$ and $n \in \mathbb{N}$. If x is $(n+1)$-exact but not n-exact, then*

$$RTZ(x, n) = x - sgn(x)2^{expo(x)-n}.$$

Proof For the case $n = 0$, we have $sig(x) = 1$ by Definition 4.2, and by Lemmas 4.1 and 6.1,

$$x - sgn(x)2^{expo(x)-n} = sgn(x)2^{expo(x)} - sgn(x)2^{expo(x)} = 0 = RTZ(x, n).$$

Thus, we may assume $n > 0$, and by Lemma 6.6, we may also assume $x > 0$.

Let $a = x - 2^{expo(x)-n}$ and $b = x + 2^{expo(x)-n}$. Since $x > 2^{expo(x)}$, $x \geq 2^{expo(x)} + 2^{expo(x)+1-n}$ by Lemma 4.20, and hence $a \geq 2^{expo(x)}$ and $expo(a) = expo(x)$. It follows that $b = fp^+(a, n)$.

By hypothesis, $x2^{n-expo(x)} \in \mathbb{Z}$ but $x2^{n-expo(x)}/2 = x2^{n-1-expo(x)} \notin \mathbb{Z}$, and therefore, $x2^{n-expo(x)}$ is odd. Let $x2^{n-expo(x)} = 2k + 1$. Then

$$a2^{n-1-expo(a)} = (x - 2^{expo(x)-n})2^{n-1-expo(x)} = (2k+1)/2 - 1/2 = k \in \mathbb{Z}.$$

Thus, a is n-exact, and by Lemma 4.19, so is $a + 2^{expo(a)+1-n} = b$. Now by Lemma 6.9, $a \leq RTZ(x, n)$, but if $a < RTZ(x, n)$, then since $RTZ(x, n)$ is n-exact, Lemma 4.20 would imply $b \leq RTZ(x, n)$, contradicting $x < b$. Therefore, $a = RTZ(x, n)$. □

Figure 6.1 is provided as a visual aid in understanding the following lemma, which formulates the conditions under which a truncated sum may be computed by truncating one of the summands in advance of the addition.

Lemma 6.14 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $k \in \mathbb{Z}$. If $x \geq 0$, $y \geq 0$, and x is $(k + expo(x) - expo(y))$-exact, then*

$$x + RTZ(y, k) = RTZ(x + y, k + expo(x + y) - expo(y)).$$

Proof Let $n = k + expo(x) - expo(y)$. Since x is n-exact,

$$x2^{k-1-expo(y)} = x2^{n-1-expo(x)} \in \mathbb{Z}.$$

$$\begin{array}{r}
\overbrace{\overbrace{\texttt{xxxxxxxxxx}}^{k}}^{k+expo(x)-expo(y)} \\
\texttt{1xxxxx.xxxxxxxxx|00}\cdots \\
+\quad \texttt{1yyy.yyyyyyyyy|yy}\cdots \\
\hline
\underbrace{\texttt{1zzzzzz.zzzzzzzzz}}_{k+expo(x+y)-expo(y)}\texttt{|zz}\cdots
\end{array}$$

Fig. 6.1 Lemma 6.14

Let $k' = k + expo(x + y) - expo(y)$. Then by Lemma 1.1,

$$\begin{aligned}
x + RTZ(y, k) &= x + \lfloor 2^{k-1-expo(y)} y \rfloor 2^{expo(y)+1-k} \\
&= (x2^{k-1-expo(y)} + \lfloor 2^{k-1-expo(y)} y \rfloor) 2^{expo(y)+1-k} \\
&= \lfloor 2^{k-1-expo(y)}(x + y) \rfloor 2^{expo(y)+1-k} \\
&= \lfloor 2^{k'-1-expo(x+y)}(x + y) \rfloor 2^{expo(x+y)+1-k'} \\
&= RTZ(x + y, k').
\end{aligned}$$

□

Lemma 6.14 holds for subtraction as well, but only if the summands are properly ordered.

Lemma 6.15 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $k \in \mathbb{Z}$. If $y > x > 0$, $k + expo(x - y) - expo(y) > 0$, and x is $(k + expo(x) - expo(y))$-exact, then*

$$x - RTZ(y, k) = RTZ(x - y, k + expo(x - y) - expo(y)).$$

Proof Let $n = k + expo(x) - expo(y)$. Since x is n-exact,

$$x2^{k-1-expo(y)} = x2^{n-1-expo(x)} \in \mathbb{Z}.$$

Let $k' = k + expo(x - y) - expo(y)$. Then by Lemma 1.1,

$$\begin{aligned}
x - RTZ(y, k) &= x - \lfloor 2^{k-1-expo(y)} y \rfloor 2^{expo(y)+1-k} \\
&= -(\lfloor 2^{k-1-expo(y)} y \rfloor - x2^{k-1-expo(y)}) 2^{expo(y)+1-k} \\
&= -\lfloor 2^{k-1-expo(y)}(y - x) \rfloor 2^{expo(y)+1-k} \\
&= -\lfloor 2^{k'-1-expo(y-x)}(y - x) \rfloor 2^{expo(y-x)+1-k'} \\
&= -RTZ(y - x, k') \\
&= RTZ(x - y, k').
\end{aligned}$$

□

Truncation of a bit vector may be described as a shifted bit slice.

Lemma 6.16 *Let $x \in \mathbb{N}$ and $k \in \mathbb{N}$, and let $n = expo(x) + 1$. If $0 < k \leq n$, then*

$$RTZ(x, k) = 2^{n-k} x[n-1 : n-k].$$

Proof By Lemmas 2.4 and 2.12,

$$\begin{aligned} RTZ(x, k) &= \lfloor 2^{k-1-expo(x)} x \rfloor 2^{expo(x)+1-k} \\ &= \lfloor x/2^{n-k} \rfloor 2^{n-k} \\ &= 2^{n-k} \lfloor x/2^{n-k} \rfloor [k-1 : 0] \\ &= 2^{n-k} x[n-1 : n-k]. \end{aligned}$$

□

Corollary 6.17 *Let $x \in \mathbb{N}$, $i \in \mathbb{N}$, $j \in \mathbb{N}$, and $k \in \mathbb{N}$. If $0 < k \leq expo(x) + 1$ and $expo(x) + 1 - k \leq j \leq i$ $expo(x)$, then*

$$RTZ(x, k)[i : j] = x[i : j]$$

Proof Let $n = expo(x) + 1$. By Lemmas 6.16, 2.14, and 2.19,

$$\begin{aligned} RTZ(x, k)[i : j] &= (2^{n-k} x[n-1 : n-k])[i : j] \\ &= x[n-1 : n-k][i+k-n : j+k-n] \\ &= x[i : j]. \end{aligned}$$

□

Corollary 6.18 *Let $x \in \mathbb{N}$, $m \in \mathbb{N}$, and $k \in \mathbb{N}$, and let $n = expo(x) + 1$. If $x \geq 0$ and $0 < k < m \leq n$, then*

$$RTZ(x, m) = RTZ(x, k) + 2^{n-m} x[n-k-1 : n-m].$$

Proof By Lemmas 6.16 and 2.18,

$$\begin{aligned} RTZ(x, m) &= 2^{n-m} x[n-1 : n-m] \\ &= 2^{n-m} (2^{m-k} x[n-1 : n-k] + x[n-k-1 : n-m]) \\ &= 2^{n-k} x[n-1 : n-k] + 2^{n-m} x[n-k-1 : n-m] \\ &= RTZ(x, k) + 2^{n-m} x[n-k-1 : n-m]. \end{aligned}$$

□

Corollary 6.19 *Let $x \in \mathbb{N}$, $m \in \mathbb{N}$, and $k \in \mathbb{N}$, and let $n = expo(x)$. If $0 < k < n \leq m$, then*

$$RTZ(x, k) = x \,\&\, (2^m - 2^{n-k}).$$

Proof By Lemmas 6.16 and 3.13,

$$RTZ(x, k) = 2^{n-k}x[n-1 : n - k] = x \mathbin{\&} (2^n - 2^{n-k}),$$

and by Lemmas 3.2 2.4, and 3.6,

$$\begin{aligned} x \mathbin{\&} (2^n - 2^{n-k}) &= \left(x \mathbin{\&} (2^n - 2^{n-k})\right)[n-1 : 0] \\ &= x[n-1 : 0] \mathbin{\&} (2^n - 2^{n-k})[n-1 : 0] \\ &= x \mathbin{\&} (2^n - 2^{n-k})[n-1 : 0]. \end{aligned}$$

But since

$$2^m - 2^{n-k} = 2^n(2^{m-n} - 1) + 2^n - 2^{n-k},$$

$$(2^m - 2^{n-k})[n-1 : 0] = (2^m - 2^{n-k}) \mathit{mod}\ 2^n = 2^n - 2^{n-k}.$$

□

6.2 Rounding Away from Zero

The dual of truncation is defined similarly, using the ceiling instead of the floor.

Definition 6.2 For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}$,

$$RAZ(x, n) = sgn(x)\lceil 2^{n-1}sig(x)\rceil 2^{expo(x)-n+1}.$$

Example Let $x = 45/8 = (101.101)_2$ and $n = 5$. Then $sgn(x) = 1$, $expo(x) = 2$, $sig(x) = (1.01101)_2$,

$$\lceil 2^{n-1}sig(x)\rceil 2^{1-n} = \lceil (10110.1)_2\rceil 2^{-4} = (10111)_2 \cdot 2^{-4} = (1.0111)_2,$$

and

$$RAZ(x, n) = \lceil 2^{n-1}sig(x)\rceil 2^{1-n}2^{expo(x)} = (1.0111)_2 \cdot 2^2 = (101.11)_2.$$

Note that this value is the smallest 5-exact number not exceeded by x.

The negative-precision case is less than intuitive.

Lemma 6.20 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}$, if $n \leq 0$, then*

$$RAZ(x, n) = sgn(x)2^{expo(x)+1-n}.$$

Proof By Lemma 4.3,

$$0 < 2^{n-1} sig(x) \leq sig(x)/2 < 1,$$

and hence, by Lemma 1.6,

$$RAZ(x,n) = sgn(x)\lceil 2^{n-1} sig(x) \rceil 2^{expo(x)+1-n} = sgn(x) 2^{expo(x)+1-n}.$$

□

We have the following bounds on $RAZ(x, n)$.

Lemma 6.21 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$,*

$$|RAZ(x,n)| \geq |x|.$$

Proof By Lemmas 1.6 and 4.1,

$$|RAZ(x,n)| \geq 2^{n-1} sig(x) 2^{expo(x)-n+1} = sig(x) 2^{expo(x)} = |x|.$$

□

Lemma 6.22 *If $x \in \mathbb{R}$, $x \neq 0$, and $n \in \mathbb{Z}^+$, then*

$$|RAZ(x,n)| < |x| + 2^{expo(x)-n+1} \leq |x|(1 + 2^{1-n}).$$

Proof By Definitions 6.2 and 1.1 and Lemma 4.1,

$$|RAZ(x,n)| < (2^{n-1} sig(x) + 1) 2^{expo(x)-n+1} = |x| + 2^{expo(x)-n+1}.$$

The second inequality follows from Definition 4.1. □

Corollary 6.23 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$,*

$$|RAZ(x,n) - x| < 2^{expo(x)-n+1} \leq 2^{1-n}|x|.$$

Proof By Lemmas 6.21,

$$|RAZ(x,n) - x| = ||RAZ(x,n)| - |x|| = |RAZ(x,n)| - |x|.$$

The corollary now follows from Lemma 6.22. □

Unlike *RTZ*, *RAZ* is not guaranteed to preserve the exponent of its argument, but the only exception is the case in which a number is rounded up to a power of 2.

Lemma 6.24 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$, if $|RAZ(x,n)| \neq 2^{expo(x)+1}$, then*

$$expo(RAZ(x,n)) = expo(x).$$

Proof By Lemma 4.3,

$$\begin{aligned}|RAZ(x,n)| &= \lceil 2^{n-1}sig(x)\rceil 2^{expo(x)-n+1}\\ &\leq \lceil 2^{n}\rceil 2^{expo(x)-n+1}\\ &= 2^{n}2^{expo(x)-n+1}\\ &= 2^{expo(x)+1}.\end{aligned}$$

The claim now follows from Lemmas 6.21 and 4.2 (c). □

The standard rounding mode properties may now be derived.

Lemma 6.25 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}$,*

$$RAZ(-x,n) = -RAZ(x,n).$$

Proof This is an immediate consequence of Definition 6.1 □

Lemma 6.26 *If $x \in \mathbb{R}$ and $n \in \mathbb{Z}^{+}$, then $RAZ(x,n)$ is n-exact.*

Proof By Corollary 6.24 and Lemma 4.10, we may assume that

$$expo(RAZ(x,n)) = expo(x).$$

Consequently, it suffices to observe that

$$RAZ(x,n)2^{n-1-expo(x)} = sgn(x)\lceil 2^{n-1}sig(x)\rceil \in \mathbb{Z}.$$

□

Lemma 6.27 *Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}^{+}$. If x is n-exact, then*

$$RAZ(x,n) = x.$$

Proof By Definition 4.2 and Lemma 1.7,

$$\lceil 2^{n-1}sig(x)\rceil = 2^{n-1}sig(x),$$

and hence by Definition 6.2 and Lemma 4.1,

$$\begin{aligned}RAZ(x,n) &= sgn(x)\lceil 2^{n-1}sig(x)\rceil 2^{expo(x)-n+1}\\ &= sgn(x)2^{n-1}sig(x)2^{expo(x)-n+1}\\ &= sgn(x)sig(x)2^{expo(x)}\\ &= x.\end{aligned}$$

□

Lemma 6.28 *Let $x \in \mathbb{R}$, $a \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If a is n-exact and $a \geq x$, then $a \geq RAZ(x, n)$.*

Proof If $x < 0$, then

$$x = -|x| \geq -|RAZ(x, n)| = RAZ(x, n)$$

by Lemma 6.21. Therefore, we may assume that $x \geq 0$. Suppose $a < RAZ(x, n)$. Then by Lemmas 4.20 and 4.2 (c),

$$RAZ(x, n) \geq a + 2^{expo(a)-n+1} \geq x + 2^{expo(x)-n+1},$$

contradicting Lemma 6.22. □

Corollary 6.29 *Let $x \in \mathbb{R}$, $a \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If a is n-exact and $a > |RTZ(x, n)|$, then $a \geq |RAZ(x, n)|$.*

Proof We may assume that $x > 0$ and x is not n-exact. By Lemma 6.9, $a > x$, and by Lemma 6.28, $a \geq RAZ(x, n)$. □

Lemma 6.30 *Let $x \in \mathbb{R}$, $k \in \mathbb{Z}$, $m \in \mathbb{Z}^+$, and $n \in \mathbb{Z}^+$, with $m < n$ and $|x| < 2^k$. If $|RAZ(x, n)| = 2^k$, then $|RAZ(x, m)| = 2^k$.*

Proof We may assume that $x > 0$ and $RAZ(x, n) = 2^k$. By Lemmas 6.21, 6.27, and 4.23,

$$RAZ(x, m) \geq x > fp^-(2^k, n) > fp^-(2^k, m).$$

By Lemmas 4.20 and 4.24, $RAZ(x, m) \geq 2^k$, and the lemma follows from Lemma 6.28. □

Lemma 6.31 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If $x \leq y$, then*

$$RAZ(x, n) \leq RAZ(y, n).$$

Proof First suppose $x > 0$. By Lemma 6.21, $RAZ(y, n) \geq y \geq x$. Since $RAZ(x, n)$ is n-exact by Lemma 6.26, Lemma 6.28 implies

$$RAZ(y, n) \geq RAZ(x, n).$$

Now suppose $x \leq 0$. We may assume that $x \leq y < 0$. Thus, since $0 < -y \leq -x$, we have $RAZ(-y, n) \leq RAZ(-x, n)$ and by Lemma 6.25,

$$RAZ(x, n) = -RAZ(-x, n) \leq -RAZ(-y, n) = RAZ(y, n).$$

□

Lemma 6.32 *For all $x \in \mathbb{R}$, $n \in \mathbb{Z}^+$, and $k \in \mathbb{Z}^+$,*

$$RAZ(2^k x, n) = 2^k RAZ(x, n).$$

Proof By Lemma 4.6,

$$\begin{aligned}RAZ(2^k x, n) &= sgn(2^k x)\lceil 2^{n-1} sig(2^k x)\rceil 2^{expo(2^k x)-n+1}\\ &= sgn(x)\lceil 2^{n-1} sig(x)\rceil 2^{expo(x)+k-n+1}\\ &= 2^k RAZ(x, n).\end{aligned}$$

□

Lemma 6.33 *Let $x \in \mathbb{R}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$. If $0 < m \leq n$, then*

$$RAZ(RAZ(x, n), m) = RAZ(x, m).$$

Proof We may assume $x > 0$. Consider first the case

$$RAZ(x, n) = 2^{expo(x)+1}.$$

In this case, $RAZ(x, n)$ is m-exact, so that

$$RAZ(RAZ(x, n), m) = RAZ(x, n) = 2^{expo(x)+1}.$$

By Lemma 6.24, we need only show that $RAZ(x, m) \geq 2^{expo(x)+1}$. But since $m \leq n$, $RAZ(x, m)$ is n-exact, and since $RAZ(x, m) \geq x$, $RAZ(x, m) \geq RAZ(x, n)$ by Lemma 6.28.

Thus, we may assume $RAZ(x, n) < 2^{expo(x)+1}$. By Corollary 6.24,

$$expo(RAZ(x, n)) = expo(x),$$

and hence by Lemma 1.8,

$$\begin{aligned}&RAZ(RAZ(x, n), m)\\ &= \lceil 2^{m-1-expo(x)}(\lceil 2^{n-1-expo(x)} x\rceil 2^{expo(x)+1-n})\rceil 2^{expo(x)+1-m}\\ &= \lceil\lceil 2^{n-1-expo(x)} x\rceil / 2^{n-m}\rceil 2^{expo(x)+1-m}\\ &= \lceil 2^{n-1-expo(x)} x / 2^{n-m}\rceil 2^{expo(x)+1-m}\\ &= \lceil 2^{m-1-expo(x)} x\rceil 2^{expo(x)+1-m}\\ &= RAZ(x, m).\end{aligned}$$

□

The next four results correspond to Lemmas 6.13, 6.12, 6.14, and 6.15 of the preceding section.

Lemma 6.34 *Let $x \in \mathbb{R}$ and $n \in \mathbb{N}$. If x is $(n + 1)$-exact but not n-exact, then*

$$RAZ(x, n) = x + sgn(x)2^{expo(x)-n}.$$

Proof For the case $n = 0$, we have $sig(x) = 1$ by Definition 4.2, and by Lemmas 4.1 and 6.20,

$$\begin{aligned} x + sgn(x)2^{expo(x)-n} &= sgn(x)2^{expo(x)} + sgn(x)2^{expo(x)} \\ &= sgn(x)2^{expo(x)+1} \\ &= RAZ(x, n). \end{aligned}$$

Thus, we may assume $n > 0$, and by Lemma 6.25, we may also assume $x > 0$. Let $a = x - 2^{expo(x)-n}$ and $b = x + 2^{expo(x)-n}$. As noted in the proof of Lemma 6.13, a and b are both n-exact and $b = fp^+(a, n)$. Now by Lemma 6.28, $b \geq RAZ(x, n)$, but if $b = fp^+(a, n) > RAZ(x, n)$, then since $RAZ(x, n)$ is n-exact, Lemma 4.20 would imply $a \geq RAZ(x, n)$, contradicting $a < x$. Therefore, $a = RAZ(x, n)$. □

Lemma 6.35 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $k \in \mathbb{Z}$. If $x \geq 0$, $y \geq 0$, and x is $(k + expo(x) - expo(y))$-exact, then*

$$x + RAZ(y, k) = RAZ(x + y, k + expo(x + y) - expo(y)).$$

Proof Let $n = k + expo(x) - expo(y)$. Since x is n-exact,

$$x2^{k-1-expo(y)} = x2^{n-1-expo(x)} \in \mathbb{Z}.$$

Let $k' = k + expo(x + y) - expo(y)$. Then by Lemma 1.7(d),

$$\begin{aligned} x + RAZ(y, k) &= x + \lceil 2^{k-1-expo(y)} y \rceil 2^{expo(y)+1-k} \\ &= (x2^{k-1-expo(y)} + \lceil 2^{k-1-expo(y)} y \rceil) 2^{expo(y)+1-k} \\ &= \lceil 2^{k-1-expo(y)}(x + y) \rceil 2^{expo(y)+1-k} \\ &= \lceil 2^{k'-1-expo(x+y)}(x + y) \rceil 2^{expo(x+y)+1-k'} \\ &= RAZ(x + y, k'). \end{aligned}$$

□

Lemma 6.36 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $k \in \mathbb{Z}$. If $x > y > 0$, $k + expo(x - y) - expo(y) > 0$, and x is $(k + expo(x) - expo(y))$-exact, then*

$$x - RTZ(y, k) = RAZ(x - y, k + expo(x - y) - expo(y)).$$

Proof Let $n = k + expo(x) - expo(y)$. Since x is n-exact,

$$x2^{k-1-expo(y)} = x2^{n-1-expo(x)} \in \mathbb{Z}.$$

Let $k' = k + expo(x - y) - expo(y)$. Then by Lemma 1.1,

$$\begin{aligned}
x - RTZ(y, k) &= x - \lfloor 2^{k-1-expo(y)} y \rfloor 2^{expo(y)+1-k} \\
&= -(\lfloor 2^{k-1-expo(y)} y \rfloor - x 2^{k-1-expo(y)}) 2^{expo(y)+1-k} \\
&= -\lfloor 2^{k-1-expo(y)} (y - x) \rfloor 2^{expo(y)+1-k} \\
&= -\lfloor 2^{k'-1-expo(y-x)} (y - x) \rfloor 2^{expo(y-x)+1-k'} \\
&= -RTZ(y - x, k') \\
&= RTZ(x - y, k').
\end{aligned}$$

□

The following result combines Lemmas 6.15 and 6.36.

Corollary 6.37 *Let $x \in \mathbb{R}$ and $y \in \mathbb{R}$ such that $x \neq 0$, $y \neq 0$, and $x + y \neq 0$. Let $k \in \mathbb{Z}$,*

$$k' = k + expo(x) - expo(y),$$

and

$$k'' = k + expo(x + y) - expo(y).$$

If $k'' > 0$, and x is k'-exact, then

$$x + RTZ(y, k) = \begin{cases} RTZ(x + y, k'') & \text{if } sgn(x + y) = sgn(y) \\ RAZ(x + y, k'') & \text{if } sgn(x + y) \neq sgn(y). \end{cases}$$

Proof By Lemmas 6.6 and 6.25, we may assume that $x > 0$. The case $y > 0$ is handled by Lemma 6.14. For the case $y < 0$, Lemmas 6.15 and 6.36 cover the subcases $-y > x$ and $-y < x$, respectively. □

We turn now to the problem of bit-level implementation of rounding. Truncation, according to Lemma 6.16, is equivalent to a bit slice operation, which may be implemented as a logical operation using Corollary 6.19. Other rounding modes may be reduced to the case of truncation by a method known as *constant injection*. Let x be m-exact with $expo(x) = e$, say

$$x = (1.\beta_1 \beta_2 \cdots \beta_{m-1})_2 \cdot 2^e,$$

to be rounded to n bits, where $n \leq m$, according to a rounding mode $\mathcal{R}$. Our goal is to construct a *rounding constant* $\mathcal{C}$, depending on m, n, e, and $\mathcal{R}$, such that

$$\mathcal{R}(x, n) = RTZ(x + \mathcal{C}, n).$$

The appropriate constant for the case $\mathcal{R} = RAZ$ is

$$\mathcal{C} = 2^e(2^{-(n-1)} - 2^{-(m-1)}) = 2^{e+1}(2^{-n} - 2^{-m}),$$

which consists of a string of 1s at the bit positions corresponding to the least significant $m - n$ bits of x, as illustrated below.

$$\begin{array}{rcccccccccl} & 1 . & \beta_1 \beta_2 & \cdots & \beta_{n-1} & \beta_n & \cdots & \beta_{m-1} & \times & 2^e \\ + & 1 . & 0 \; 0 & \cdots & 0 & 1 & \cdots & 1 & \times & 2^e \\ \hline \end{array}$$

As suggested by the diagram, the addition $x + \mathcal{C}$ generates a carry into the position of β_{n-1} unless $\beta_n = \cdots = \beta_{m-1} = 0$, i.e., unless x is n-exact, and x is rounded up accordingly. This observation is formalized by the following lemma.

Lemma 6.38 *Let $x \in \mathbb{R}$, $m \in \mathbb{Z}^+$, and $n \in \mathbb{Z}^+$. If x is m-exact, $x > 0$, and $m \geq n$, then*

$$RAZ(x, n) = RTZ(x + 2^{expo(x)+1}(2^{-n} - 2^{-m}), n).$$

Proof Let $a = RTZ(x + 2^{expo(x)+1}(2^{-n} - 2^{-m}), n)$. Since a and $RAZ(x, n)$ are both n-exact and

$$a < x + 2^{expo(x)+1-n} \leq RAZ(x, n) + 2^{expo(RAZ(x,n))+1-n},$$

$a \leq RAZ(x, n)$ by Lemma 4.20.

If x is n-exact, then $a \geq RTZ(x, n) = x = RAZ(x, n)$, and hence $a = RAZ(x, n)$. Thus, we may assume x is not n-exact. But then since $x > RTZ(x, n)$ and x is m-exact,

$$x \geq RTZ(x, n) + 2^{expo(x)+1-m}$$

and hence

$$x + 2^{expo(x)+1}(2^{-n} - 2^{-m}) \geq RTZ(x, n) + 2^{expo(x)+1-n} = RAZ(x, n),$$

which implies $a \geq RAZ(x, n)$. □

6.3 Rounding to Nearest

Next, we examine the mode *RNE*, which may round in either direction, selecting the representable number that is closest to its argument. This mode is known as "rounding to nearest even" because of the manner in which it resolves the

ambiguous case of a number that is equidistant from two successive representable numbers.

Definition 6.3 Given $x \in \mathbb{R}$ and $n \in \mathbb{Z}$, let $z = \lfloor 2^{n-1} sig(x) \rfloor$, and $f = 2^{n-1} sig(x) - z$. Then

$$RNE(x, n) = \begin{cases} RTZ(x, n) \text{ if } f < 1/2 \text{ or } f = 1/2 \text{ and } z \text{ is even} \\ RAZ(x, n) \text{ if } f > 1/2 \text{ or } f = 1/2 \text{ and } z \text{ is odd.} \end{cases}$$

Example Let $x = (101.101)_2$ and $n = 5$. Then

$$z = \lfloor 2^{n-1} sig(x) \rfloor = \lfloor (10110.1)_2 \rfloor = (10110)_2$$

and

$$f = 2^{n-1} sig(x) - z = (10110.1)_2 - (10110)_2 = (0.1)_2 = 1/2,$$

indicating a "tie", i.e., that x is equidistant from two successive 5-exact numbers. Since z is even, the tie is broken in favor of the lesser of the two:

$$RNE(x, n) = RTZ(x, n) = (101.10)_2.$$

Like all rounding modes, the value of *RNE* is always that of either *RTZ* or *RAZ*. We list several properties of *RNE* that may be derived from this observation and the corresponding properties of *RTZ* and *RAZ*.

Lemma 6.39 *If $x \in \mathbb{R}$, $n \in \mathbb{Z}^+$, then $RNE(x, n)$ is n-exact.*

Lemma 6.40 *Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$. If x is n-exact, then*

$$RNE(x, n) = x.$$

Lemma 6.41 *Let $x \in \mathbb{R}$, $a \in \mathbb{R}$, and $n \in \mathbb{N}$. Suppose a is n-exact.*

(a) If $a \geq x$, then $a \geq RNE(x, n)$;
(b) If $a \leq x$, then $a \leq RNE(x, n)$.

Lemma 6.42 *For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$, if $|RNE(x, n)| \neq 2^{expo(x)+1}$, then*

$$expo(RNE(x, n)) = expo(x).$$

Lemma 6.43 *For all $x \in \mathbb{R}$, $n \in \mathbb{Z}^+$, and $k \in \mathbb{Z}$,*

$$RNE(2^k x, n) = 2^k RNE(x, n).$$

Proof It is clear from Definition 6.3 that the choice between $RTZ(x, n)$ and $RAZ(x, n)$ depends only on $sig(x)$. Thus, for example, if $RNE(x, n) = RTZ(x, n)$,

then since $sig(2^k x) = sig(x)$, $RNE(2^k x, n) = RTZ(2^k x, n)$ as well, and by Lemma 6.11,

$$RNE(2^k x, n) = RTZ(2^k x, n) = 2^k RTZ(x, n) = 2^k RNE(x, n).$$

□

Lemma 6.44 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}$,*

$$RNE(-x, n) = -RNE(x, n).$$

Proof This may be derived from Lemmas 6.6 and 6.25 by following the same reasoning as used in the proof of Lemma 6.43. □

In the computation of $RNE(x, n)$, the choice between $RTZ(x, n)$ and $RAZ(x, n)$ is governed by their relative distances from x.

Lemma 6.45 *Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$.*

(a) If $|x - RTZ(x, n)| < |x - RAZ(x, n)|$, then $RNE(x, n) = RTZ(x, n)$.
(b) If $|x - RTZ(x, n)| > |x - RAZ(x, n)|$, then $RNE(x, n) = RAZ(x, n)$.

Proof We may assume that $2^{n-1} sig(x) \notin \mathbb{Z}$, for otherwise

$$RTZ(x, n) = RAZ(x, n) = RNE(x, n) = x.$$

Let $f = 2^{n-1} sig(x) - \lfloor 2^{n-1} sig(x) \rfloor$. Then

$$\begin{aligned} |x - RTZ(x, n)| &= |x| - |RTZ(x, n)| \\ &= 2^{expo(x)+1-n}(2^{n-1} sig(x) - \lfloor 2^{n-1} sig(x) \rfloor) \\ &= 2^{expo(x)+1-n} f \end{aligned}$$

and

$$\begin{aligned} |x - RAZ(x, n)| &= |RAZ(x, n)| - |x| \\ &= 2^{expo(x)+1-n}(\lceil 2^{n-1} sig(x) \rceil - 2^{n-1} sig(x)) \\ &= 2^{expo(x)+1-n}(1 - f). \end{aligned}$$

Thus, (a) and (b) correspond to $f < 1/2$ and $f > 1/2$, respectively. □

No n-exact number can be closer to x than is $RNE(x, n)$:

Lemma 6.46 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If y is n-exact, then*

$$|x - y| \geq |x - RNE(x, n)|.$$

Proof Assume $|x - y| < |x - RNE(x, n)|$. We shall only consider the case $x > 0$, as the case $x < 0$ is handled similarly.

First suppose $RNE(x, n) = RTZ(x, n)$. Since $RNE(x, n) \le x$, we must have $y > RNE(x, n)$ and hence $y > x$ by Lemma 6.9. But since $RAZ(x, n) - x \ge x - RNE(x, n)$ by Lemma 6.45, we also have $y < RAZ(x, n)$, and hence $y < x$ by Lemma 6.28.

In the remaining case, $RNE(x, n) = RAZ(x, n) > x$. Now $y < RNE(x, n)$ and by Lemma 6.28, $y < x$. But in this case, Lemma 6.45 implies $y > RTZ(x, n)$, and hence $y > x$ by Lemma 6.9. □

Consequently, the maximum *RNE* rounding error is half the distance between successive representable numbers.

Lemma 6.47 *If $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$, then*

$$|x - RNE(x, n)| \le 2^{expo(x)-n} \le 2^{-n}|x|.$$

Proof By Lemma 6.44, we may assume $x > 0$. Let

$$a = RTZ(x, n) + 2^{expo(x)+1-n} = fp^+(RTZ(x, n), n).$$

If the statement fails, then since $RTZ(x, n)$ and $RAZ(x, n)$ are both n-exact, Lemma 6.46 implies

$$RTZ(x, n) < x - 2^{expo(x)-n} < x + 2^{expo(x)-n} < RAZ(x, n),$$

and hence $a < RAZ(x, n)$. Then by Lemmas 4.20 and 6.28, we have $a < x$, contradicting Lemma 6.4. □

Corollary 6.48 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If $x \ne 0$, y is n-exact, and $|x - y| < 2^{expo(x)-n}$, then $y = RNE(x, n)$.*

Proof We shall consider the case $x > 0$; the case $x < 0$ then follows from Lemma 6.44.

Let $e = expo(x)$. and $z = RNE(x, n)$. By Lemma 6.42, $expo(z) \ge e$. We also have $expo(y) \ge e$, for otherwise $y < 2^e$ and by Lemma 4.25,

$$y \le fp^-(2^e, n) = 2^e - 2^{e-n} \le x - 2^{e-n},$$

contradicting $|x - y| < 2^{e-n}$. By Lemma 6.47,

$$|y - z| \le |x - y| + |x - z| < 2^{e-n} + 2^{e-n} = 2^{e+1-n}.$$

If $y < z$, then since

$$z < y + 2^{e+1-n} \le y + 2^{expo(y)+1-n} = fp^+(y, n),$$

Lemma 4.20 implies $y = z$. But if $z < y$, we have a similar contradiction. □

The one remaining rounding mode axiom, monotonicity, may also be derived as a consequence of Lemma 6.46:

Lemma 6.49 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If $x \leq y$, then*

$$RNE(x,n) \leq RNE(y,n).$$

Proof Suppose $x < y$ and $RNE(x,n) > RNE(y,n)$. Then Lemma 6.46 implies $x > RNE(y,n)$, otherwise $x \leq RNE(y,n) < RNE(x,n)$ and $|x - RNE(y,n)| < |x - RNE(x,n)|$. Similarly, $y < RNE(x,n)$, and thus $RNE(y,n) < x < y < RNE(x,n)$. Applying Lemma 6.46 again, we have $x - RNE(y,n) \geq RNE(x,n) - x$, and hence $2x \geq RNE(x,n) + RNE(y,n)$. Similarly, $RNE(x,n) - y \geq y - RNE(y,n)$, and hence $RNE(x,n) + RNE(y,n) \geq 2y$. Consequently, $2x \geq 2y$, contradicting $x < y$. □

We have the following analog of Lemma 6.30:

Lemma 6.50 *Let $x \in \mathbb{R}$, $k \in \mathbb{N}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$, with $0 < m < n$ and $|x| < 2^k$. If $|RNE(x,n)| = 2^k$, then $|RNE(x,m)| = 2^k$.*

Proof We may assume that $x > 0$. By Lemmas 4.23 and 6.41,

$$x > fp^-(2^k,n) = 2^k - 2^{k-n} \geq 2^k - 2^{k-m-1} > 2^k - 2^{k-m} = fp^-(2^k,m).$$

Let $y = 2^k - 2^{k-m-1} = 2^{k-1}(2 - 2^{-m})$. Then

$$2^{m-1} sig(y) = 2^{m-1}(2 - 2^{-m}) = 2^m - \frac{1}{2}.$$

If $z = \lfloor 2^{m-1} sig(y) \rfloor = 2^m - 1$ and $f = 2^{m-1} sig(y) - z = \frac{1}{2}$, then according to Definition 6.3, since z is odd,

$$RNE(y,m) = RAZ(y,m) \geq y > fp^-(2^k,m).$$

By Lemma 6.49,

$$RNE(x,m) \geq RNE(y,m) \geq y > fp^-(2^k,m),$$

which implies $RNE(x,m) = 2^k$. □

A *midpoint* with respect to a precision n may be characterized as a number that is $(n+1)$-exact but not n-exact. By virtue of the following result, the term *rounding boundary* is sometimes used as well.

Lemma 6.51 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{N}$. If $0 < x < y$ and $RNE(x,n) \neq RNE(y,n)$, then for some $a \in \mathbb{R}$, $x \leq a \leq y$ and a is $(n+1)$-exact.*

Proof By Lemma 6.49, $RNE(x,n) < RNE(y,n)$. Let $e = expo(RNE(x,n))$,

$$a = fp^+(RNE(x,n), n+1) = RNE(x,n) + 2^{e-n}$$

and

$$b = fp^+(RNE(x,n), n) = RNE(x,n) + 2^{e+1-n} = a + 2^{e-n}.$$

Since $RNE(x,n)$ and $RNE(y,n)$ are both n-exact by Lemma 6.39, Lemma 4.19 implies that a is $(n+1)$-exact and b is n-exact and consequently, by Lemma 4.20, a is not n-exact and $RNE(y,n) \geq b$. Thus,

$$RNE(x,n) < a < b \leq RNE(y,n).$$

Moreover, $x \leq a \leq y$, for if $x > a$, then $|x - b| < |x - RNE(x,n|$, contradicting Lemma 6.46 and similarly if $y < a$, then $|y - RNE(x,n)| < |y - RNE(y,n|$. □

Lemma 6.52 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, $a \in \mathbb{R}$, $n \in \mathbb{Z}^+$, and $k \in \mathbb{Z}^+$ with $n \geq k$. If a is $(n+1)$-exact, $0 < a < x$, and $0 < y < a + 2^{expo(a)-n}$, then*

$$RNE(x,k) \geq RNE(y,k).$$

Proof By Lemma 6.49, we may assume $x < y$, so that $a < x < y < a+2^{expo(a)-n}$. By Lemmas 4.19 and 4.20, a and $a+2^{expo(a)-n}$ are successive $(n+1)$-exact numbers, and hence $RNE(x,k) = RNE(y,k)$ by Lemma 6.51. □

We also have the following partial converse of Lemma 6.51.

Lemma 6.53 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, $a \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If $0 < x < a < y$ and a is $(n+1)$-exact but not n-exact, then*

$$RNE(x,n) < RNE(y,n).$$

Proof Let $e = expo(a)$. Since a is not n-exact, $a \neq 2^e$. Let

$$b = fp^-(a, n+1) = a - 2^{e-n}.$$

By Lemma 4.24,

$$fp^+(b, n+1) = b + 2^{expo(b)-n} = a = b + 2^{e-n},$$

and it follows that $expo(b) = e$.

By hypothesis, $2^{n-e}a \in \mathbb{Z}$ but $2^{n-1-e}a \notin \mathbb{Z}$, and therefore, $2^{n-e}a$ is odd, i.e., $2^{n-e}a = 2k+1$, where $k \in \mathbb{Z}$. Now since

$$2^{n-1-e}b = 2^{n-1-e}(a - 2^{n-e}) = \frac{1}{2}(2^{n-e}a - 1) = k \in \mathbb{Z},$$

b is n-exact. Let

$$c = fp^+(b, n) = b + 2^{n+1-e} = a + 2^{n-e}.$$

By Lemma 4.19, c is n-exact as well.

If $RNE(x, n) > b$, then by Lemma 4.20, $RNE(x, n) \geq c$, which implies $|b - x| < |RNE(x, n) - x|$, contradicting Lemma 6.46. Similarly, if $RNE(y, n) < c$, then $RNE(y, n) \leq b$ and $|c - y| < |RNE(y, n) - y|$. Thus,

$$RNE(x, n) \leq b < a < c \leq RNE(y, n).$$

□

The meaning of "round to nearest even" is that in the case of a midpoint x, $RNE(x, n)$ is defined to be the "rounder" of the two nearest n-exact numbers, i.e., the one that is $(n - 1)$-exact.

Lemma 6.54 *Let $n \in \mathbb{N}$, $n > 1$, and $x \in \mathbb{R}$. If x is $(n + 1)$-exact but not n-exact, then $RNE(x, n)$ is $(n - 1)$-exact.*

Proof Again we may assume $x > 0$. Let $z = \lfloor 2^{n-1} sig(x) \rfloor$ and $f = 2^{n-1} sig(x) - z$. Since $2^{n-1} sig(x) \notin \mathbb{Z}$, $0 < f < 1$. But $2^n sig(x) = 2z + 2f \in \mathbb{Z}$, hence $2f \in \mathbb{Z}$ and $f = \frac{1}{2}$.

If z is even, then

$$RNE(x, n) = RTZ(x, n) = z2^{expo(x)+1-n}$$

and by Lemma 6.2,

$$2^{n-2-expo(RNE(x,n))} RNE(x, n) = 2^{n-2-expo(x)} z 2^{expo(x)+1-n} = z/2 \in \mathbb{Z}.$$

If z is odd, then

$$RNE(x, n) = RAZ(x, n) = (z + 1)2^{expo(x)+1-n}.$$

We may assume $RAZ(x, n) \neq 2^{expo(x)+1}$, and hence by Corollary 6.24,

$$\begin{aligned} 2^{n-2-expo(RNE(x,n))} RNE(x, n) &= 2^{n-2-expo(x)} (z + 1) 2^{expo(x)+1-n} \\ &= (z + 1)/2 \\ &\in \mathbb{Z}. \end{aligned}$$

□

As we have noted, RNE does not satisfy the property of decomposability:

Example Let $x = 43 = (101011)_2$.

$$RNE(RNE(x, 5), 3) = RNE((101100)_2, 3) = (110000)_2 = 48,$$

whereas

$$RNE(x, 3) = (101000)_2 = 40.$$

A consequence of Lemma 6.54 is that a midpoint is sometimes rounded up and sometimes down, and therefore, over the course of a long series of computations and approximations, rounding error is less likely to accumulate to a significant degree in one direction than it would be if the choice were made more consistently.

Intuitively, a rounding mode $\mathcal{R}$ is *unbiased* if the expected error that it incurs in rounding a representative set of values to a given precision is 0. Given a rounding precision $n > 1$, one choice of "representative set" is the set of all m-exact numbers for some $m > n$. In order to limit the error computation to finite sets, we seek to identify a partition of the real numbers such that for all $m > n$, the expected error over the m-exact numbers within each interval of the partition may be shown to be 0. In practice, we find that a suitable partition is the set of $(n-1)$-exact numbers. If we assume that $\mathcal{R}$ is symmetric, then we need only consider positive values. Thus, we consider the set of m-exact numbers that lie between two consecutive $(n-1)$-exact values $x_0 > 0$ and $fp^+(x_0, n-1) = x_0 + 2^{expo(x_0)+2-n}$, i.e.,

$$\{x_0 + 2^{expo(x_0)+1-m}k \mid 0 \leq k < 2^{n+1-m}\}.$$

This leads to the following definition:

Definition 6.4 A symmetric rounding mode $\mathcal{R}$ is *unbiased* if for all $n \in N$, $m \in N$, and $x_0 \in R$, if $m > n > 1$, $x_0 > 0$, and x_0 is $(n-1)$-exact, then the expected error

$$\mathcal{E}(\mathcal{R}, n, m, x_0) = \frac{1}{2^{m+1-n}} \sum_{k=0}^{2^{m+1-n}-1} (\mathcal{R}(x_k, n) - x_k) = 0,$$

where $x_k = x_0 + 2^{expo(x_0)+1-m}k$ for $0 \leq k < 2^{m+1-n}$.

This property holds for $\mathcal{R} = RNE$:

Lemma 6.55 *RNE is an unbiased rounding mode.*

Proof Let $e = expo(x_0)$, where $x_0 > 0$ and x_0 is $(n-1)$-exact, and let $N = 2^{m+1-n}$. For $0 \leq k \leq N$, let $x_k = x_0 + 2^{e+1-m}k$ and $\epsilon_k = \mathcal{R}(x_k, n) - x_k$. Since x_0 and $x_{N/2}$ are both n-exact, $\epsilon_0 = \epsilon_{N/2} = 0$ and the sum in Definition 6.4 may be expressed as

$$\mathcal{E}(\mathcal{R}, n, m, x_0) = \frac{1}{2^{n+1-m}}(\Sigma_1 + \epsilon_{N/4} + \Sigma_2 + \Sigma_3 + \epsilon_{3N/4} + \Sigma_4),$$

where

$$\Sigma_1 = \sum_{k=1}^{\frac{N}{4}-1} \epsilon_k, \quad \Sigma_2 = \sum_{k=\frac{N}{4}+1}^{\frac{N}{2}-1} \epsilon_k, \quad \Sigma_3 = \sum_{k=\frac{N}{2}}^{\frac{3N}{4}-1} \epsilon_k, \quad \text{and } \Sigma_4 = \sum_{k=\frac{3N}{4}+1}^{N-1} \epsilon_k.$$

It follows easily from Lemmas 4.19 and 4.20 that x_N is $(n-1)$-exact, $x_{N/2}$ is n-exact but not $(n-1)$-exact, $x_{N/4}$ and $x_{3N/4}$ are $(n+1)$-exact but not n-exact, and that no other n-exact numbers lie between x_0 and x_N. By Lemmas 6.39, 6.47, and 6.54, $RNE(x_{N/4}, n) = x_0$ and $RNE(x_{3N/4}, n) = x_N$, and hence

$$\epsilon_{N/4} + \epsilon_{3N/4} = (x_0 - x_{N/4}) + (x_N - x_{3N/4}) = -2^{e-n} + 2^{e-n} = 0.$$

By Corollary 6.48, $RNE(x_k, n) = x_0$ for $0 < k < N/4$, and $RNE(x_k, n) = x_{N/2}$ for $N/4 < k < N/2$. Therefore,

$$\Sigma_1 = \sum_{k=1}^{\frac{N}{4}-1} (x_0 - x_k) = \sum_{k=1}^{\frac{N}{4}-1} (-2^{e+1-m} k) = -2^{e+1-m} \sum_{k=1}^{\frac{N}{4}-1} k$$

and

$$\begin{aligned}
\Sigma_2 &= \sum_{k=\frac{N}{4}+1}^{\frac{N}{2}} (x_{N/2} - x_k) \\
&= \sum_{k=\frac{N}{4}+1}^{\frac{N}{2}} \big((x_{N/2} - x_0) + (x_0 - x_k)\big) \\
&= \frac{N}{4}(2^{e+1-n}) - 2^{e+1-m} \sum_{k=\frac{N}{4}+1}^{\frac{N}{2}} k \\
&= 2^{e+1-m} \left(\frac{N}{4}\frac{N}{2} - \sum_{k=\frac{N}{4}+1}^{\frac{N}{2}} k \right).
\end{aligned}$$

Thus,

$$\begin{aligned}
\Sigma_1 + \Sigma_2 &= 2^{e+1-m} \left(-\sum_{k=1}^{\frac{N}{4}-1} k + \frac{N}{4}\frac{N}{2} - \sum_{k=\frac{N}{4}+1}^{\frac{N}{2}} k \right) \\
&= 2^{e+1-m} \left(\frac{N^2}{8} + \frac{N}{4} - \sum_{k=1}^{\frac{N}{2}} k \right) \\
&= 2^{e+1-m} \left(\frac{N^2}{8} + \frac{N}{4} - \frac{1}{2}\frac{N}{2}\left(\frac{N}{2} + 1\right) \right) \\
&= 0.
\end{aligned}$$

Similarly, $\Sigma_3 + \Sigma_4 - 0$ and the lemma follows. □

The cost of this feature is a complicated definition requiring an expensive implementation. When the goal of a computation is provable accuracy rather than IEEE-compliance, a simpler version of "round to nearest" may be appropriate. The critical feature of this mode then becomes the relative error bound guaranteed by Lemma 6.47, since this is likely to be the basis for any formal error analysis. The following definition presents an alternative to *RNE* known as "round to nearest away from zero". This variant, which is mentioned but not prescribed by the standard, respects the same error bound as *RNE* (see Lemma 6.66) and admits a simpler implementation, and is therefore commonly used for internal floating-point calculations.

Definition 6.5 Given $x \in \mathbb{R}$ and $n \in \mathbb{Z}$, let $z = \lfloor 2^{n-1} sig(x) \rfloor$, and $f = 2^{n-1} sig(x) - z$. Then

$$RNA(x, n) = \begin{cases} RTZ(x, n) & \text{if } f < 1/2 \\ RAZ(x, n) & \text{if } f \geq 1/2. \end{cases}$$

Example Let $x = (101.101)_2$ and $n = 5$. Since

$$f = 2^{n-1} sig(x) - \lfloor 2^{n-1} sig(x) \rfloor = (10110.1)_2 - (10110)_2 = (0.1 = 1/2)_2,$$

$$RNA(x, n) = RAZ(x, n) = (101.11)_2.$$

Naturally, many of the properties of *RNE* are held by *RNA* as well. We list some of them here, omitting the proofs, which are essentially the same as those given above for *RNE*.

Lemma 6.56 *For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$, $RNA(x, n)$ is n-exact.*

Lemma 6.57 *Let $x \in \mathbb{R}$ and $n \in \mathbb{N}$. If x is n-exact, then*

$$RNA(x, n) = x.$$

Lemma 6.58 *Let $x \in \mathbb{R}$, $a \in \mathbb{R}$, and $n \in \mathbb{N}$. Suppose a is n-exact.*

(a) If $a \geq x$, then $a \geq RNA(x, n)$;
(b) If $a \leq x$, then $a \leq RNA(x, n)$.

Lemma 6.59 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If $x \leq y$, then*

$$RNA(x, n) \leq RNA(y, n).$$

Lemma 6.60 *For all $x \in \mathbb{R}$, $n \in \mathbb{Z}^+$, and $k \in \mathbb{Z}$,*

$$RNA(2^k x, n) = 2^k RNA(x, n).$$

Lemma 6.61 *For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$, if $|RNA(x, n)| \neq 2^{expo(x)+1}$, then*

$$expo(RNA(x, n)) = expo(x).$$

Lemma 6.62 *For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}$,*

$$RNA(-x, n) = -RNA(x, n).$$

Lemma 6.63 *Let $x \in \mathbb{R}$ and $n \in \mathbb{N}$.*

(a) If $|x - RTZ(x, n)| < |x - RAZ(x, n)|$, then $RNA(x, n) = RTZ(x, n)$.
(b) If $|x - RTZ(x, n)| > |x - RAZ(x, n)|$, then $RNA(x, n) = RAZ(x, n)$.

Lemma 6.64 *Let $x \in \mathbb{R}$, $k \in \mathbb{N}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$. $0 < m < n$, $|x| < 2^k$, and $|RNA(x, n)| = 2^k$, then $|RNE(x, m)| = 2^k$.*

Lemma 6.65 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. If y is n-exact, then*

$$|x - y| \geq |x - RNA(x, n)|.$$

Lemma 6.66 *If $x \in \mathbb{R}$, $n \in \mathbb{Z}^+$, then*

$$|x - RNA(x, n)| \leq 2^{expo(x)-n}.$$

Lemma 6.67 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{N}$. If $0 < x < y$ and $RNA(x, n) \neq RNA(y, n)$, then for some $a \in \mathbb{R}$, $x \leq a \leq y$ and a is $(n + 1)$-exact.*

Lemma 6.68 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, $a \in \mathbb{R}$, $n \in \mathbb{Z}^+$, and $k \in \mathbb{Z}^+$ with $n \geq k$. If a is $(n+1)$-exact, $0 < a < x$, and $0 < y < a+2^{expo(a)-n}$, then $RNA(x, k) \geq RNA(y, k)$.*

The difference between *RNE* and *RNA* is that the latter always rounds a midpoint away from 0.

Lemma 6.69 *Let $x \in \mathbb{R}$ and $n \in \mathbb{N}$. If x is $(n + 1)$-exact but not n-exact, then*

$$RNA(x, n) = RAZ(x, n) = x + sgn(x)2^{expo(x)-n}.$$

Proof By Lemmas 6.44 and 6.25, we may assume $x > 0$. Let $z = \lfloor 2^{n-1} sig(x) \rfloor$ and $f = 2^{n-1} sig(x) - z$. Since $2^{n-1} sig(x) \notin \mathbb{Z}$, $0 < f < 1$. But $2^n sig(x) = 2z + 2f \in \mathbb{Z}$, hence $2f \in \mathbb{Z}$ and $f = \frac{1}{2}$. Therefore, according to Definition 6.5, $RNA(x, n) = RAZ(x, n)$. The second inequality is a restatement of Lemma 6.34.

□

RNA is not decomposable, as illustrated by the same example that we used for *RNE*:

Example Let $x = 43 = (101011)_2$.

$$RNA(RNA(x, 5), 3) = RNA((101100)_2, 3) = (110000)_2 = 48,$$

whereas

$$(RNA(x, 3) = (101000)_2 = 40.$$

There is one case of a midpoint for which *RNE* and *RNA* are guaranteed to produce the same result: if the greater of the two representable numbers that are equidistant from x is a power of 2, i.e., $x = 2^{expo(x)+1} - 2^{expo(x)-n}$, then both modes round to this number.

Lemma 6.70 *Let $n \in \mathbb{N}$, $n > 1$, and $x \in \mathbb{R}$, $x > 0$. If $x + 2^{expo(x)-n} \geq 2^{expo(x)+1}$, then*

$$RNE(x, n) = RNA(x, n) = 2^{expo(x)+1} = RTZ(x + 2^{expo(x)-n}, n).$$

Proof Suppose $RNE(x, n) \neq 2^{expo(x)+1}$. Then Lemma 6.42 implies

$$RNE(x, n) < 2^{expo(x)+1}$$

and by Lemmas 4.20 and 6.47,

$$\begin{aligned} 2^{expo(x)+1} &\geq RNE(x, n) + 2^{expo(x)+1-n} \\ &\geq x - 2^{expo(x)-n} + 2^{expo(x)+1-n} \\ &= x + 2^{expo(x)-n} \\ &\geq 2^{expo(x)+1}. \end{aligned}$$

It follows that $x = 2^{expo(x)+1} - 2^{expo(x)-n}$, which is easily seen to be $(n+1)$-exact but not n-exact, while $RNE(x, n) = 2^{expo(x)+1} - 2^{expo(x)+1-n}$ is n-exact but not $(n-1)$-exact, contradicting Lemma 6.54.

Now suppose $RNA(x, n) \neq 2^{expo(x)+1}$. Using Lemmas 6.61 and 6.66, we may show in the same way as above that $x = 2^{expo(x)+1} - 2^{expo(x)-n}$. Once again, x is $(n+1)$-exact but not n-exact, and hence, by Lemma 6.69,

$$RNA(x, n) = x + 2^{expo(x)-n} = 2^{expo(x)+1},$$

a contradiction.

Finally, suppose $2^{expo(x)+1} \neq RTZ(x+2^{expo(x)-n}, n)$. Since $2^{expo(x)+1}$ is n-exact, $2^{expo(x)+1} < RTZ(x + 2^{expo(x)-n}, n)$ by Lemma 6.9. But then by Lemma 4.20,

$$RTZ(x + 2^{expo(x)-n}, n) \geq 2^{expo(x)+1} + 2^{expo(x)+2-n} > x + 2^{expo(x)-n},$$

contradicting Lemma 6.3. □

The additive property shared by *RTZ* and *RAZ* that is described in Lemmas 6.14 and 6.35, respectively, does not hold for *RNE* in precisely the same form.

Example Let $x = 2 = (10)_2$, $y = 5 = (101)_2$, and $k = 2$. Then

$$k + expo(x) - expo(y) = 2 + 1 - 2 = 1$$

and

$$k + expo(x + y) - expo(y) = 2 + 2 - 2 = 2.$$

Although x is clearly $(k + expo(x) - expo(y))$-exact,

$$x + RNE(y, k) = (10)_2 + RNE((101)_2, 2) = (10)_2 + (100)_2 = (110)_2,$$

while

$$RNE(x + y, k + expo(x + y) - expo(y)) = RNE((111, 2)_2) = (1000)_2.$$

However, this property is shared by *RNA*, and a slightly weaker version holds for *RNE*.

Lemma 6.71 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $k \in \mathbb{Z}$ with $x \geq 0$ and $y \geq 0$. Let*

$$k' = k + expo(x) - expo(y)$$

and

$$k'' = k + expo(x + y) - expo(y).$$

(a) If x is $(k' - 1)$-exact, then

$$x + RNE(y, k) = RNE(x + y, k'').$$

(b) If x is k'-exact, then

$$x + RNA(y, k) = RNA(x + y, k'').$$

Proof

(a) Applying Lemmas 6.14 and 6.35, we need only show that either

$$RNE(y, k) = RTZ(y, k) \text{ and } RNE(x + y, k'') = RTZ(x + y, k'')$$

or

$$RNE(y, k) = RAZ(y, k) \text{ and } RNE(x + y, k'') = RAZ(x + y, k'').$$

Let $z_1 = \lfloor 2^{k-1} sig(y) \rfloor$, $f_1 = 2^{k-1} sig(y) - z_1$, $z_2 = \lfloor 2^{k''-1} sig(x+y) \rfloor$, and $f_2 = 2^{k''-1} sig(x + y) - z_2$. According to Definition 6.3, it will suffice to show that

$$2^{k''-1} sig(x + y) - 2^{k-1} sig(y) = 2\ell,$$

for some $\ell \in \mathbb{Z}$, for then Lemma 1.1 will imply that $z_2 = z_1 + 2\ell$ and $f_2 = f_1$. But

$$\begin{aligned} 2^{k''-1}sig(x+y) - 2^{k-1}sig(y) &= 2^{k''-expo(x+y)-1}(x+y) - 2^{k-expo(y)-1}y \\ &= 2^{k-expo(y)-1}(x+y) - 2^{k-expo(y)-1}y \\ &= 2^{k-expo(y)-1}x \\ &= 2^{k'-expo(x)-1}x \\ &= 2\ell, \end{aligned}$$

where

$$\ell = 2^{(k'-1)-expo(x)-1} \in \mathbb{Z}$$

by Lemma 4.8.

(b) Here we must show that either

$$RNA(y,k) = RTZ(y,k) \text{ and } RNA(x+y,k'') = RTZ(x+y,k'')$$

or

$$RNA(y,k) = RAZ(y,k) \text{ and } RNA(x+y,k'') = RAZ(x+y,k'').$$

According to Definition 6.5, this is true whenever $f_1 = f_2$. Thus, we need only show that

$$2^{k''-1}sig(x+y) - 2^{k-1}sig(y) = 2^{k'-expo(x)-1}x \in \mathbb{Z},$$

which is equivalent to the hypothesis that x is k'-exact. □

The rounding constant $\mathcal{C}$ (see the discussion preceding Lemma 6.38) for both of the modes *RNE* and *RNA* is a simple power of 2, equal to half the value of the least significant bit of the rounded result. That is, if the rounding precision is n and the unrounded result is

$$x = (1.\beta_1\beta_2\cdots)_2 \times 2^e,$$

then $\mathcal{C} = 2^{e-n}$, as illustrated below.

$$\begin{array}{rl} & 1\,.\,\beta_1\beta_2\cdots\ \beta_{n-1}\beta_n\cdots \times\ 2^e \\ + & 1\,.\,0\ 0\ \cdots\ 0\ 1\qquad \times\ 2^e \\ \hline \end{array}$$

The following lemma exposes the extra expense of implementing *RNE* as compared to *RNA*. While the correctly rounded result is given by $RTZ(x + \mathcal{C}, n)$ in most cases, special attention is required for the computation of *RNE* in the case where it differs from *RNA*, i.e., when x is $(n+1)$-exact and $\beta_n = 1$. In this case, the least significant bit must be forced to 0. This is accomplished by truncating $x + \mathcal{C}$ to $n - 1$ bits rather than n.

Lemma 6.72 *If* $n \in \mathbb{N}$, $n > 1$, $x \in \mathbb{R}$, *and* $x > 0$, *then*

$$RNE(x,n) = \begin{cases} RTZ(x+2^{expo(x)-n}, n-1) & \text{if } x \text{ is } (n+1)\text{-exact but not } n\text{-exact} \\ RTZ(x+2^{expo(x)-n}, n) & \text{otherwise} \end{cases}$$

and

$$RNA(x,n) = RTZ(x + 2^{expo(x)-n}, n).$$

Proof If $x + 2^{expo(x)-n} \geq 2^{expo(x)+1}$, then by Lemma 6.70,

$$RNE(x,n) = RNA(x,n) = 2^{expo(x)+1} = RTZ(x + 2^{expo(x)-n}, n).$$

But then, by Lemmas 6.12, 4.10, and 6.8,

$$\begin{aligned} RTZ(x + 2^{expo(x)-n}, n-1) &= RTZ(RTZ(x + 2^{expo(x)-n}, n), n-1) \\ &= RTZ(2^{expo(x)+1}, n-1) \\ &= 2^{expo(x)+1} \\ &= RTZ(x + 2^{expo(x)-n}, n). \end{aligned}$$

Thus, we may assume $x + 2^{expo(x)-n} < 2^{expo(x)+1}$, and it follows from Lemmas 6.42, 6.47, 6.61, and 6.66 that

$$expo(RNE(x,n)) = expo(RNA(x,n)) = expo(x + 2^{expo(x)-n}) = expo(x).$$

Case 1: x is n-exact

By Lemma 6.9, $rtz(x + 2^{expo(x)-n}, n) \geq x$. But since

$$RTZ(x + 2^{expo(x)-n}, n) \leq x + 2^{expo(x)-n} < x + 2^{expo(x)+1-n},$$

Lemma 4.20 yields $RTZ(x + 2^{expo(x)-n}, n) \leq x$, and hence

$$RTZ(x + 2^{expo(x)-n}, n) = x = RNE(x,n) = RNA(x,n).$$

Case 2: x is not $(n+1)$-exact

We have $RNE(x,n) > x - 2^{expo(x)-n}$, for otherwise Lemma 6.47 <would imply

$$RNE(x,n) = x - 2^{expo(x)-n},$$

and since $RNE(x, n)$ is $(n + 1)$-exact, so would be

$$RNE(x, n) + 2^{expo(RNE(x,n))-n} = x - 2^{expo(x)-n} + 2^{expo(RNE(x,n))-n} = x.$$

Since $RNE(x, n) \leq x + 2^{expo(x)-n}$, $RNE(x, n) \leq RTZ(x + 2^{expo(x)-n}, n)$ by Lemma 6.9. But since

$$RTZ(x + 2^{expo(x)-n}, n) \leq x + 2^{expo(x)-n} < RNE(x, n) + 2^{expo(x)+1-n},$$

$RTZ(x + 2^{expo(x)-n}, n) \leq RNE(x, n)$.

The same argument applies to $RNA(x, n)$, but with Lemma 6.66 invoked in place of Lemma 6.47.

Case 3: x is $(n + 1)$-exact but not n-exact

The identity for $RNA(x, n)$ is given by Lemma 6.69. To prove the claim for RNE, we first consider the case $RNE(x, n) > x$. Since $RNE(x, n)$ is $(n + 1)$-exact, $RNE(x, n) \geq x+2^{expo(x)-n}$, hence $RNE(x, n) = x+2^{expo(x)-n}$, and by Lemma 6.54,

$$RTZ(x + 2^{expo(x)-n}, n - 1) = RTZ(RNE(x, n), n - 1) = RNE(x, n).$$

Now suppose $RNE(x, n) < x$. Then $RNE(x, n) < x + 2^{expo(x)-n}$ implies $RNE(x, n) \leq RTZ(x + 2^{expo(x)-n}, n - 1)$. But since

$$\begin{aligned} RTZ(x + 2^{expo(x)-n}, n - 1) &\leq x + 2^{expo(x)-n} \\ &= x - 2^{expo(x)-n} + 2^{expo(x)+1-n} \\ &< RNE(x, n) + 2^{expo(x)+2-n}, \end{aligned}$$

we have $RTZ(x + 2^{expo(x)-n}, n - 1) \leq RNE(x, n)$. □

As a consequence of the preceding lemma, $RNA(x, m)$ depends only on the most significant $m + 1$ bits of x.

Lemma 6.73 *Let $x \in \mathbb{R}$, $m \in \mathbb{Z}^+$, and $n \in \mathbb{Z}^+$. If $n > m$, then*

$$RNA(RTZ(x, n), m) = RNA(x, m).$$

Proof By Lemmas 6.62 and 6.6, we may assume that $x > 0$. Furthermore, it will suffice to consider the case $n = m + 1$, because then for $n > m + 1$,

$$\begin{aligned} RNA(RTZ(x, n), m) &= RNA(RTZ(RTZ(x, n), m + 1), m) \\ &= RNA(RTZ(x, m + 1), m) \\ &= RNA(x, m). \end{aligned}$$

Thus, according to Lemmas 6.72 and 6.2, our goal is to prove

$$RTZ(RTZ(x, m + 1) + 2^{expo(x)-n}, m) = RTZ(x + 2^{expo(x)-n}, m),$$

but after applying Lemma 6.3, we need only show

$$RTZ(RTZ(x, m+1) + 2^{expo(x)-n}, m) \geq RTZ(x + 2^{expo(x)-n}, m).$$

Let

$$y = RTZ(x, m+1) + 2^{expo(x)-n} = fp^+(RTZ(x, m+1), m+1).$$

By Lemmas 6.7 and 4.19, y is $(m+1)$-exact. Now since $RTZ(x + 2^{expo(x)-n}, m)$ is also $(m+1)$-exact and Lemmas 6.3 and 6.4 imply

$$\begin{aligned} RTZ(x + 2^{expo(x)-n}, m) &\leq x + 2^{expo(x)-n} \\ &< RTZ(x, m+1) + 2^{expo(x)-n} + 2^{expo(x)-n} \\ &= y + 2^{expo(x)-n} \\ &= fp^+(y, m+1), \end{aligned}$$

Lemma 4.20 yields $RTZ(x + 2^{expo(x)-n}, m) \leq y$. Finally, by Lemma 6.9,

$$RTZ(x + 2^{expo(x)-n}, m) \leq RTZ(y, m).$$

□

6.4 Odd Rounding

A landmark paper of 1946 by von Neumann et al. contains an early discussion of rounding [3][1]:

> . . . the round-off is intended to produce satisfactory n-digit approximations for the product xy and the quotient x/y of two n-digit numbers. Two things are wanted of the round-off: (1) The approximation should be good, i.e., its variance from the "true" xy or x/y should be as small as practical; (2) The approximation should be unbiased, i.e., its mean should be equal to the "true" xy or x/y.

The authors are willing to relax the exclusion of bias, noting that it generally incurs a cost in efficiency and concluding that "we shall not complicate the machine by introducing such corrections." Two methods of rounding are recommended:

> The first class is characterized by its ignoring all digits beyond the nth, and even the nth digit itself, which it replaces by a 1. The second class is characterized by the procedure of adding one unit in the $n+1^{st}$ digit, performing the carries which this may induce, and then keeping only the first n digits.

[1] A notable aspect of this paper, which is often cited as the "birth certificate of computer science", is its position, expounded in Section 5.3, that floating-point arithmetic is generally a bad idea.

The second of these is the mode that we have designated *RNA* in the preceding section. The first, which has come to be known as *von Neumann rounding*, may be formulated as

$$\mathcal{R}(x, n) = RTZ(x, n-1) + sgn(x)2^{expo(x)+1-n}.$$

This mode has twice the error range of rounding to nearest but is much easier to implement, involving neither carry propagation nor operand analysis. On the other hand, not only is it slightly biased, but it is in violation of our second axiom, since $\mathcal{R}(x, n) \neq x$ when x is $(n-1)$-exact. Just as modern computing has replaced *RNA* with the less efficient but strictly unbiased *RNE*, we have the following variant of von Neumann rounding, known as *sticky* [22] or *odd* [1] rounding:

Definition 6.6 If $x \in \mathbb{R}$, $n \in \mathbb{N}$, and $n > 1$, then

$$RTO(x, n) = \begin{cases} x & \text{if } x \text{ is } (n-1)\text{-exact} \\ RTZ(x, n-1) + sgn(x)2^{expo(x)+1-n} & \text{otherwise.} \end{cases}$$

We shall see that odd rounding holds all of the properties discussed at the beginning of this chapter, as well as others that are relevant to the double rounding problem (Lemma 6.84) and the implementation of floating-point addition (Lemma 6.85).

Lemma 6.74 *Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$. If x is n-exact, then*

$$RTO(x, n) = x.$$

Proof We may assume that x is not $(n-1)$-exact, and hence Lemma 6.13 yields

$$RTZ(x, n-1) = x - sgn(x)2^{expo(x)+1-n}.$$

Thus,

$$RTO(x, n) = RTZ(x, n-1) + sgn(x)2^{expo(x)+1-n} = x.$$

□

It follows that *RTO* is symmetric:

Lemma 6.75 *For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$,*

$$RTO(-x, n) = -RTO(x, n).$$

Proof If x is $(n-1)$-exact, then so is $-x$, and

$$RTO(-x, n) = -x = -RTO(x, n).$$

Otherwise, by Lemmas 6.6 and 4.5,

$$\begin{aligned} RTO(-x, n) &= RTZ(-x, n-1) + sgn(-x)2^{expo(-x)+1-n} \\ &= -RTZ(x, n-1) - sgn(x)2^{expo(x)+1-n} \\ &= -RTO(x, n). \end{aligned}$$

□

Definition 6.4 may now be applied:

Lemma 6.76 *RTO is an unbiased rounding mode.*

Proof Let x_0, e, n, m, N, and x_k be defined as in the proof of Lemma 6.55. Then for $0 < k < N$, $x_0 < x_k < x_N$, where x_0 and x_N are successive $(n-1)$-exact numbers, and therefore

$$RTO(x_k, n) = RTZ(x_k, n-1) + 2^{expo(x_k)+1-n} = x_0 + 2^{e+1-n} = x_{N/2}.$$

Thus,

$$\begin{aligned} \sum_{k=0}^{N-1}(RTO(x_k, n) - x_k) &= \sum_{k=1}^{N-1}(x_{N/2} - x_k) \\ &= \sum_{k=1}^{N-1}(x_{N/2} - x_0 - 2^{e+1-m}k) \\ &= (N-1)(x_{N/2} - x_0) - 2^{e+1-m}\sum_{k=1}^{N-1} k \\ &= (N-1)2^{e+1-n} - 2^{e+1-m} \cdot \frac{N}{2}(N-1) \\ &= (N-1)(2^{e+1-n} - 2^{e+1-m} \cdot 2^{m-n}) \\ &= 0. \end{aligned}$$

□

Lemma 6.77 *If $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$, then*

$$expo(RTO(x, n)) = expo(x).$$

Proof By Lemma 6.75, we may assume that $x > 0$. If x is $(n-1)$-exact, the claim is trivial. Suppose x is not $(n-1)$-exact. By Lemma 6.2, $RTZ(x, n-1) < 2^{expo(x)+1}$. Since $RTZ(x, n-1)$ and $2^{expo(x)+1}$ are both $(n-1)$-exact, Lemma 4.20 implies

$$\begin{aligned} 2^{expo(x)+1} &\geq fp^{+}(RTZ(x, n-1), n-1) \\ &= RTZ(x, n-1) + 2^{expo(x)+1-(n-1)} \\ &> RTZ(x, n-1) + 2^{expo(x)+1-n} \\ &= RTO(x, n), \end{aligned}$$

and it follows that $expo(RTO(x, n)) = expo(x)$. □

Lemma 6.78 *For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$, $RTO(x, n)$ is n-exact.*

Proof If x is $(n-1)$-exact, then $RTO(x, n) = x$. Suppose x is not $(n-1)$-exact. Since $RTZ(x, n-1)$ is $(n-1)$-exact, i.e.,

$$2^{(n-1)-1-expo(RTZ(x,n-1))}RTZ(x, n-1) = 2^{n-2-expo(x)}RTZ(x, n-1) \in \mathbb{Z},$$

Lemma 6.77 implies

$$\begin{aligned} &2^{n-1-expo(RTO(x,n))}RTO(x, n) \\ &= 2^{n-1-expo(x)}(RTZ(x, n-1) + sgn(x)2^{expo(x)+1-n} \\ &= 2^{n-1-expo(x)}RTZ(x, n-1) + sgn(x) \\ &= 2 \cdot 2^{n-2-expo(x)}RTZ(x, n-1) + sgn(x) \\ &\in \mathbb{Z}. \end{aligned}$$

□

Lemma 6.79 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{N}$. If $x \leq y$, then*

$$RTO(x, n) \leq RTO(y, n).$$

Proof Using Lemma 6.75, we may assume that $0 < x < y$. Suppose first that x is $(n-1)$-exact. If y is also $(n-1)$-exact, then the claim is trivial, and if not, then by Lemmas 6.8 and 6.10,

$$\begin{aligned} RTO(x, n) &= x \\ &= RTZ(x, n-1) \\ &\leq RTZ(y, n-1) \\ &< RTZ(y, n-1) + 2^{expo(y)+1-n} \\ &= RTO(y, n). \end{aligned}$$

Similarly, if neither x nor y is $(n-1)$-exact, then Lemmas 6.10 and 4.2 (c) imply

$$\begin{aligned} RTO(x, n) &= RTZ(x, n-1)2^{expo(x)+1-n} \\ &\le RTZ(y, n-1) \\ &< RTZ(y, n-1) + 2^{expo(y)+1-n} \\ &= RTO(y, n). \end{aligned}$$

In the remaining case, y is $(n-1)$-exact and x is not. Now by Lemmas 6.7 and 4.9, $RTZ(x, n-1)$ and y are both n-exact, and hence, by Lemmas 4.20 and 6.2,

$$\begin{aligned} RTO(y, n) &= y \\ &\ge fp^+(RTZ(x, n-1), n) \\ &= RTZ(x, n-1) + 2^{expo(x)+1-n} \\ &= RTO(x, n). \end{aligned}$$

□

Lemma 6.80 *For all $x \in \mathbb{R}$, $n \in \mathbb{Z}^+$, and $k \in \mathbb{Z}$,*

$$RTO(2^k x, n) = 2^k RTO(x, n).$$

Proof If x is $(n-1)$-exact, then so is $-x$, and

$$RTO(2^k x, n) = 2^k x = 2^k RTO(x, n).$$

Otherwise, by Lemmas 6.11 and 4.6,

$$\begin{aligned} RTO(2^k x, n) &= RTZ(2^k x, n-1) + sgn(2^k x)2^{expo(2^k x)+1-n} \\ &= 2^k RTZ(x, n-1) + 2^k sgn(x)2^{expo(x)+1-n} \\ &= 2^k RTO(x, n). \end{aligned}$$

□

Lemma 6.81 *Let $m \in \mathbb{N}$, $n \in \mathbb{N}$, and $x \in \mathbb{R}$. If $n \ge m > 1$, then*

$$RTO(RTO(x, n), m) = RTO(x, m).$$

Proof If x is $(n-1)$-exact, then the claim follows trivially from Definition 6.6.

Suppose x is not $(n-1)$-exact. We may assume $x > 0$. Let $a = RTZ(x, n-1)$,

$$e = expo(x) = expo(a),$$

and

$$z = RTO(x, n) = a + 2^{e+1-n}.$$

Since a is $(n-1)$-exact and

$$a < z < a + 2^{e+2-n} = fp^+(a, n-1),$$

$RTZ(z, n-1) = a \neq z$. It follows that $expo(z) = e$ and z is not $(n-1)$-exact, which implies z is not $(m-1)$-exact. Thus,

$$\begin{aligned} RTO(RTO(x, n), m) &= RTO(z, m) \\ &= RTZ(z, m-1) + 2^{e+1-m} \\ &= RTZ(x, m-1) + 2^{e+1-m} \\ &= RTO(x, m). \end{aligned}$$

□

The following property, which is not shared by any of the other modes that we have considered, is critical to this mode's utility.

Lemma 6.82 *Let $x \in \mathbb{R}$, $m \in \mathbb{Z}^+$, and $n \in \mathbb{Z}^+$. If $n > m$, then $RTO(x, n)$ is m-exact if and only if x is m-exact.*

Proof According to Lemma 6.75, we may assume that $x > 0$. But clearly we may also assume that x is not $(n-1)$-exact, and hence, by Lemma 4.9, x is not m-exact. On the other hand, $RTZ(x, n-1)$ is $(n-1)$-exact, and since

$$\begin{aligned} RTO(x, n) &= RTZ(x, n-1) + 2^{expo(x)+1-n} \\ &< RTZ(x, n-1) + 2^{expo(x)+1-(n-1)} \\ &= fp^+(RTZ(x, n-1), n-1), \end{aligned}$$

Lemma 4.20 implies that $RTO(x, n)$ is not $(n-1)$-exact. Applying Lemma 4.9 again, we conclude that $RTO(x, n)$ is not m-exact. □

An important consequence of Lemma 6.82 is a generalization of Lemma 6.81: *a rounded result with respect to any of the modes considered thus far may be derived from an intermediate odd-rounded value*. In particular, for directed rounding, an m-bit rounded result may be derived from an $(m+1)$-bit odd rounding:

Lemma 6.83 *Let $m \in \mathbb{Z}^+$, $n \in \mathbb{Z}^+$, and $x \in \mathbb{R}$. If $n > m$, then*

$$RTZ(RTO(x, n), m) = RTZ(x, m)$$

and

$$RAZ(RTO(x, n), m) = RAZ(x, m).$$

Proof We may assume that $x > 0$ and x is not $(n-1)$-exact; the other cases follow trivially. First, note that by Lemmas 6.78 and 6.82, $RTO(x, n)$ is n-exact but not $(n-1)$-exact, and therefore, according to Lemmas 6.13 and 6.77,

$$\begin{aligned} RTZ(RTO(x, n), n-1) &= RTO(x, n) - 2^{expo(RTO(x,n))-(n-1)} \\ &= RTO(x, n) - 2^{expo(x)+1-n} \\ &= RTZ(x, n-1). \end{aligned}$$

Thus, by Lemma 6.12, for any $m < n$,

$$RTZ(RTO(x, n), m) = RTZ(RTZ(x, n-1), m) = RTZ(x, m).$$

The corresponding result for *RAZ* may be similarly derived. □

For rounding to nearest, one extra bit is required:

Lemma 6.84 *Let $m \in \mathbb{Z}^+$, $n \in \mathbb{Z}^+$, and $x \in \mathbb{R}$. If $n > m + 1$, then*

$$RNE(RTO(x, n), m) = RNE(x, m)$$

and

$$RNA(RTO(x, n), m) = RNA(x, m).$$

Proof The second equation follows easily from Lemmas 6.73 and 6.83:

$$\begin{aligned} RNA(RTO(x, n), m) &= RNA(RTZ(RTO(x, n), m+1), m) \\ &= RNA(RTZ(x, m+1), m) \\ &= RNA(x, m). \end{aligned}$$

To prove the first equation using Lemma 6.83, it will suffice to show that if $RTZ(x, m+1) = RTZ(y, m+1)$ and $RAZ(x, m+1) = RAZ(y, m+1)$, then $RNE(x, m) = RNE(y, m)$. Without loss of generality, we may assume $x \leq y$. Suppose $RNE(x, m) \neq RNE(y, m)$. Then by Lemma 6.51, for some $(m+1)$-exact a, $x \leq a \leq y$. But this implies $x = a$, for otherwise $RTZ(x, m+1) \leq x < a \leq RTZ(y, m+1)$. Similarly, $y = a$, for otherwise $RAZ(x, m+1) \leq a < y \leq RAZ(y, m+1)$. Thus, $x = y$, a contradiction. □

The following analog of Lemma 6.14 first appeared in [22]. This property is essential for the implementation of floating-point addition, as it allows a rounded sum or difference of unaligned numbers to be derived without computing the full sum explicitly. Figure 6.2 is provided as a visual aid to the proof. Note the minor departure from Fig. 6.1.

$k{+}expo(x){-}expo(y)$

k

```
  1xxxxx.xxxxxxxxx0|00 ···
+    1yyy.yyyyyyyyy|yy ···
 1zzzzzz.zzzzzzzzz|zz ···
```

$k{+}expo(x{+}y){-}expo(y)$

Fig. 6.2 Lemma 6.85

Lemma 6.85 *Let* $x \in \mathbb{R}$ *and* $y \in \mathbb{R}$ *such that* $x \neq 0$, $y \neq 0$, *and* $x + y \neq 0$. *Let* $k \in \mathbb{N}$,

$$k' = k + expo(x) - expo(y),$$

and

$$k'' = k + expo(x + y) - expo(y).$$

If $k > 1$, $k' > 1$, $k'' > 1$, *and* x *is* $(k' - 1)$*-exact, then*

$$x + RTO(y, k) = RTO(x + y, k'').$$

Proof Since x is $(k' - 1)$-exact,

$$2^{k-2-expo(y)}x = 2^{(k'-1)-1-expo(x)}x \in \mathbb{Z}.$$

Thus,

$$\begin{aligned} y \text{ is } (k-1)\text{-exact} &\Leftrightarrow 2^{k-2-expo(y)}y \in \mathbb{Z} \\ &\Leftrightarrow 2^{k-2-expo(y)}y + 2^{k-2-expo(y)}x \in \mathbb{Z} \\ &\Leftrightarrow 2^{k''-2-expo(x+y)}(x+y) \in \mathbb{Z} \\ &\Leftrightarrow x + y \text{ is } (k''-1)\text{-exact.} \end{aligned}$$

If y is $(k - 1)$-exact, then

$$x + RTO(y, k) = x + y = RTO(x + y, k'').$$

Thus, we may assume that y is not $(k - 1)$-exact. We invoke Corollary 6.37:

$$x + RTZ(y, k) = \begin{cases} RTZ(x + y, k'') \text{ if } sgn(x + y) = sgn(y) \\ RAZ(x + y, k'') \text{ if } sgn(x + y) \neq sgn(y). \end{cases}$$

Now if $sgn(x + y) = sgn(y)$, then

$$\begin{aligned} x + RTO(y, k) &= x + RTZ(y, k-1) + sgn(y)2^{expo(y)+1-k} \\ &= RTZ(x + y, k'' - 1) + sgn(x + y)2^{expo(x+y)+1-k''} \\ &= RTO(x + y, k''). \end{aligned}$$

On the other hand, if $sgn(x + y) \neq sgn(y)$, then

$$\begin{aligned} x + RTO(y, k) &= x + RTZ(y, k-1) + sgn(y)2^{expo(y)+1-k} \\ &= RAZ(x + y, k'' - 1) - sgn(x + y)2^{expo(x+y)+1-k''} \\ &= RTZ(x + y, k'' - 1) + sgn(x + y)2^{expo(x+y)+1-k''} \\ &= RTO(x + y, k''). \end{aligned}$$

□

6.5 IEEE Rounding

The IEEE standard prescribes four rounding modes: "round to nearest even" (*RNE*), "round toward 0" (*RTZ*), "round toward $+\infty$', and "round toward $-\infty$". The last two are formalized here by the following functions:

Definition 6.7 For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$,

$$RUP(x, n) = \begin{cases} RAZ(x, n) & \text{if } x \geq 0 \\ RTZ(x, n) & \text{if } x < 0. \end{cases}$$

Definition 6.8 For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$,

$$RDN(x, n) = \begin{cases} RTZ(x, n) & \text{if } x \geq 0 \\ RAZ(x, n) & \text{if } x < 0. \end{cases}$$

The essential properties of these modes are given by the following:

Lemma 6.86 *For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$,*

$$RUP(x, n) \geq x$$

and

$$RDN(x, n) \leq x.$$

Proof If $x \geq 0$, then

$$RUP(x, n) = RAZ(x, n) = |RAZ(x, n)| \geq |x| = x$$

and

$$RDN(x, n) = RTZ(x, n) = |RTZ(x, n)| \leq |x| = x;$$

if $x < 0$, then

$$RUP(x, n) = RTZ(x, n) = -|RTZ(x, n)| \geq -|x| = x$$

and

$$RUP(x, n) = RAZ(x, n) = -|RAZ(x, n)| \leq -|x| = x.$$

□

In this section, we collect a set of general results that pertain to all of the IEEE rounding modes, many of which are essentially restatements of lemmas that are proved in earlier sections. Since these results also hold for two of the other modes that we have discussed, *RAZ* and *RNA*, we shall state them as generally as possible. For this purpose, we make the following definition:

Definition 6.9 The *common rounding modes* are *RTZ*, *RAZ*, *RNE*, *RNA*, *RUP*, and *RDN*.

Note that *RUP* and *RDN* do not share the symmetry property held by the other modes. A generalization is given by the following lemma.

Lemma 6.87 *Let $\mathcal{R}$ be a common rounding mode and let*

$$\hat{\mathcal{R}} = \begin{cases} RDN & \text{if } \mathcal{R} = RUP \\ RUP & \text{if } \mathcal{R} = RDN \\ \mathcal{R} & \text{otherwise.} \end{cases}$$

For all $x \in \mathbb{R}$ and $n \in \mathbb{Z}$,

$$\mathcal{R}(-x, n) = -\hat{\mathcal{R}}(x, n).$$

Proof Suppose, for example, that $\mathcal{R} = RUP$ and $x > 0$. Then since $-x < 0$,

$$\mathcal{R}(-x, n) = RUP(-x, n) = RTZ(-x, n) = -RTZ(x, n) = -\hat{\mathcal{R}}(x, n).$$

The other cases are handled similarly. □

Lemma 6.88 *Let $\mathcal{R}$ be a common rounding mode. For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$, either*

$$\mathcal{R}(x, n) = RTZ(x, n)$$

or

$$\mathcal{R}(x, n) = RAZ(x, n).$$

Proof This is an immediate consequence of Definitions 6.3, 6.5, 6.7, and 6.8. □

Most of following results are consequences of Lemma 6.88 and the corresponding lemmas pertaining to *RTZ* and *RAZ*.

Lemma 6.89 *Let $\mathcal{R}$ be a common rounding mode. For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$, $\mathcal{R}(x, n)$ is n-exact.*

Lemma 6.90 *Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$ and let $\mathcal{R}$ be a common rounding mode. If x is n-exact, then*

$$\mathcal{R}(x, n) = x.$$

Lemma 6.91 *Let $x \in \mathbb{R}$, $a \in \mathbb{R}$, and $n \in \mathbb{Z}^+$, and let $\mathcal{R}$ be a common rounding mode. Suppose a is n-exact.*

(a) If $a \geq x$, then $a \geq \mathcal{R}(x, n)$;
(b) If $a \leq x$, then $a \leq \mathcal{R}(x, n)$.

Lemma 6.92 *If $x \in \mathbb{R}$, $n \in \mathbb{Z}^+$, and $\mathcal{R}$ is a common rounding mode, then*

$$sgn(\mathcal{R}(x, n)) = sgn(x).$$

Lemma 6.93 *Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$ and let $\mathcal{R}$ be a common rounding mode. Then*

$$|x - \mathcal{R}(x, n)| < 2^{expo(x)-n+1}.$$

Lemma 6.94 *Let $\mathcal{R}$ be a common rounding mode. For all $x \in \mathbb{R}$ and $n \in \mathbb{N}$, if $|\mathcal{R}(x, n)| \neq 2^{expo(x)+1}$, then*

$$expo(\mathcal{R}(x, n)) = expo(x).$$

Lemma 6.95 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}^+$. Let $\mathcal{R}$ be a common rounding mode. If $x \leq y$, then*

$$\mathcal{R}(x, n) \leq \mathcal{R}(y, n).$$

Proof The modes *RTZ*, *RAZ*, *RNE*, and *RNA* are covered by Lemmas 6.10, 6.31, 6.49, and 6.59. For the modes *RUP* and *RDN*, we may assume, using

Lemma 6.92, that x and y are either both positive or both negative. It follows that either

$$\mathcal{R}(x, n) = RTZ(x, n) \leq RTZ(y, n) = \mathcal{R}(y, n)$$

or

$$\mathcal{R}(x, n) = RAZ(x, n) \leq RAZ(y, n) = \mathcal{R}(y, n).$$

□

Lemma 6.96 *Let $\mathcal{R}$ be a common rounding mode. For all $x \in \mathbb{R}$, $n \in \mathbb{N}$, and $k \in \mathbb{Z}$,*

$$\mathcal{R}(2^k x, n) = 2^k \mathcal{R}(x, n).$$

Proof The modes *RTZ*, *RAZ*, *RNE*, and *RNA* are covered by Lemmas 6.11, 6.32, 6.43, and 6.60. For the modes *RUP* and *RDN*, we may assume, using Lemma 6.87, that $x > 0$, in which case

$$RUP(2^k x, n) = RAZ(2^k x, n) = 2^k RAZ(x, n) = 2^k RUP(x, n)$$

and

$$RDN(2^k x, n) = RTZ(2^k x, n) = 2^k RTZ(x, n) = 2^k RDN(x, n).$$

□

Lemma 6.97 *Let $\mathcal{R}$ be a common rounding mode, $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $k \in \mathbb{Z}$ with $x \geq 0$ and $y \geq 0$. Let $k' = k + expo(x) - expo(y)$ and $k'' = k + expo(x + y) - expo(y)$. If x is $(k' - 1)$-exact, then*

$$x + \mathcal{R}(y, k) = \mathcal{R}(x + y, k + expo(x + y) - expo(y)).$$

Proof See Lemmas 6.14, 6.35, and 6.71. □

Lemma 6.98 *Let $x \in \mathbb{R}$ and $n \in \mathbb{N}$ and let $\mathcal{R}$ be a common rounding mode. If $x \geq 0$, then*

$$RTZ(x, n) \leq \mathcal{R}(x, n) \leq RAZ(x, n).$$

Proof This is an immediate consequence of Lemmas 6.3 and 6.21:

$$RTZ(x, n) = |RTZ(x, n)| \leq |x| \leq |RAZ(x, n)| = RAZ(x, n).$$

□

If x is not $(n+1)$-exact and y is sufficiently close to x, then x and y round to the same value under any common rounding mode:

Lemma 6.99 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $n \in \mathbb{Z}^+$ with $x > 0$. Let $\mathcal{R}$ be a common rounding mode. Assume that x is not $(n+1)$-exact. If either*

$$RTZ(x, n+1) < y < x$$

or

$$RAZ(x, n+1) > y > x,$$

then $\mathcal{R}(x, n) = \mathcal{R}(y, n)$.

Proof We show the proof for the claim pertaining to RTZ; the other is similar.

By Lemma 6.95, we need only show that $\mathcal{R}(x, n) \leq \mathcal{R}(y, n)$. We also note that by Lemmas 6.12 and 6.3,

$$RTZ(x, n) = RTZ(RTZ(x, n+1), n) \leq RTZ(x, n+1) < y.$$

Case 1: $\mathcal{R} = RTZ$ or $\mathcal{R} = RDN$

by Lemma 6.7, $RTZ(x, n)$ is n-exact. By Lemma 6.9 (with $RTZ(x, n)$ and y substituted for a and x) and Definition 6.8,

$$\mathcal{R}(x, n) = RTZ(x, n) \leq RTZ(y, n) = \mathcal{R}(y, n).$$

Case 2: $\mathcal{R} = RAZ$ or $\mathcal{R} = RUP$

By Lemma 6.33,

$$RTZ(x, n) < y < x \leq RAZ(x, n+1) \leq RAZ(RAZ(x, n+1), n) = RAZ(x, n).$$

By Definition 6.7 and Lemmas 6.26, 6.28, and 6.29,

$$\mathcal{R}(y, n) = RAZ(y, n) = RAZ(x, n) = \mathcal{R}(x, n).$$

Case 3: $\mathcal{R} = RNE$ or $\mathcal{R} = RNA$

By Lemma 6.21,

$$RTZ(x, n+1) < y < x < RAZ(x, n+1) = fp^+(RTZ(x, n+1)).$$

The claim follows from Lemmas 6.29, 6.52 and 6.68. □

Corollary 6.100 *Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$ with $x > 0$. Assume that x is not $(n+1)$-exact. There exists $\epsilon \in \mathbb{R}$, $\epsilon > 0$, such that for all $y \in \mathbb{R}$ and every common rounding mode $\mathcal{R}$, if $|x - y| < \epsilon$, then $\mathcal{R}(x, n) = \mathcal{R}(y, n)$.*

Proof According to Lemma 6.99, this holds for $\epsilon = min(x - RTZ(x, n + 1), RAZ(x, n + 1) - x)$. □

Lemma 6.101 *Let $\mathcal{R}$ be a common rounding mode, $m \in \mathbb{Z}^+$, $n \in \mathbb{Z}^+$, and $x \in \mathbb{R}$. If $n \geq m + 2$, then*

$$\mathcal{R}(x, m) = \mathcal{R}(RTO(x, n), m).$$

Proof The modes *RTZ* and *RAZ* are handled by Lemma 6.83; *RNE* and *RNA* are covered by Lemma 6.84; *RUP* and *RDN* are easily reduced to *RTZ* and *RAZ*. □

Lemma 6.102 *Let $\mathcal{R}$ be a common rounding mode, $x \in \mathbb{R}$, $k \in \mathbb{N}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}$, with $0 < m < n$ and $|x| < 2^k$. If $|rnd(x, \mathcal{R}, n)| = 2^k$, then $|rnd(x, \mathcal{R}, m)| = 2^k$.*

Proof This is a consequence of Lemmas 6.30, 6.50, 6.64, and 6.3. □

The remaining results of this section pertain to the implementation of rounding. For the sake of simplicity, our characterization of the method of constant injection is formulated in the context of bit vector rounding.

Lemma 6.103 *Let $\mathcal{R}$ be a common rounding mode, $n \in \mathbb{N}$, $n > 1$, and $x \in \mathbb{N}$ with $expo(x) \geq n$. Then*

$$\mathcal{R}(x, n) = RTZ(x + \mathcal{C}, \nu),$$

where

$$\mathcal{C} = \begin{cases} 2^{expo(x)-n} & \text{if } \mathcal{R} = RNE \text{ or } \mathcal{R} = RNA \\ 2^{expo(x)-n+1} - 1 & \text{if } \mathcal{R} = RUP \text{ or } \mathcal{R} = RAZ \\ 0 & \text{if } \mathcal{R} = RTZ \text{ or } \mathcal{R} = RDN \end{cases}$$

and

$$\nu = \begin{cases} n - 1 & \text{if } \mathcal{R} = RNE \text{ and } x \text{ is } (n+1)\text{-exact but not } n\text{-exact} \\ n & \text{otherwise.} \end{cases}$$

Proof For the modes *RAZ* and *RUP*, the identity follows from Lemma 6.38, with $m = expo(x) + 1$, and Lemma 4.11. For *RNE* and *RNA*, it reduces to Lemmas 6.72 and 6.72. For *RTZ* and *RDN*, the lemma is trivial. □

Another common implementation of rounding is provided by the following result. Suppose our objective is a correct n-bit rounding (with respect to some common rounding mode) of a precise result $z > 0$, and that we have a bit vector representation of $x = \lfloor z \rfloor$. Then the desired result may be derived by rounding x in the direction specified by the following lemma. (This result is applied repeatedly in the correctness proofs of Part V.)

Lemma 6.104 *Let $\mathcal{R}$ be a common rounding mode, $z \in \mathbb{R}$, and $n \in \mathbb{N}$. Assume that $0 < n$ and $2^n \leq z$. Let $x = \lfloor z \rfloor$ and $e = expo(x)$.*

(a) z is n-exact iff $x[e-n:0] = 0$ and $z \in \mathbb{Z}$.
(b) If any of the following conditions holds, then $\mathcal{R}(z,n) = fp^+(RTZ(x,n),n)$, and otherwise $\mathcal{R}(z,n) = RTZ(x,n)$:

- *$\mathcal{R} = RUP$ or $\mathcal{R} = RAZ$ and at least one of the following is true:*
  - *$x[e-n:0] \neq 0$;*
  - *$x \neq z$.*
- *$\mathcal{R} = RNA$ and $x[e-n] = 1$*
- *$\mathcal{R} = RNE$, $x[e-n] = 1$, and at least one of the following is true:*
  - *$x[e-n-1:0] \neq 0$;*
  - *$x \neq z$;*
  - *$x[e+1-n] = 1$;*

Proof It is clear that $expo(z) = e$. Definition 6.1,

$$RTZ(z, e+1) = \lfloor 2^{(e+1)-1} sig(z) \rfloor 2^{e-(e+1)+1} = \lfloor 2^e sig(z) \rfloor = \lfloor z \rfloor = x,$$

and hence, by Lemma 6.12, $RTZ(z,n) = RTZ(x,n)$.

(a) According to Lemmas 6.7 and 6.8, it suffices to show that $z \neq RTZ(z,n)$ iff either $x[e-n:0] \neq 0$ or $x \neq z$. By Lemma 6.3,

$$RTZ(x,n) \leq x = RTZ(z,k) \leq z,$$

and it is clear that $z \neq RTZ(z,n)$ iff either $x - RTZ(x,n) > 0$ or $x \neq z$. Now since x is k-exact, Lemma 4.9 implies that x is $(e+1)$-exact, and therefore, by Lemmas 6.7 and 6.18,

$$x - RTZ(x,n) = RTZ(x, e+1) - RTZ(x,n) = x[e-n:0].$$

(b) Since $RDN(z,n) = RTZ(z,n)$ and $RUP(z,n) = RAZ(z,n)$, we need only consider RAZ, RNE, and RNA.

Suppose $\mathcal{R} = RAZ$. By Lemmas 6.29, 6.7, and 6.26,

$$RAZ(z,n) = \begin{cases} RTZ(x,n) & \text{if } z = RTZ(z,n) \\ fp^+(RTZ(x,n),n) & \text{if } z \neq RTZ(z,n). \end{cases}$$

Thus, $RAZ(z,n) = RTZ(x,n)$ iff $z = RTZ(z,n)$. But we have shown that this holds iff $x[e-n:0] = 0$ and $z \in \mathbb{Z}$, as stated by the lemma.

For the remaining cases, RNE, and RNA, we refer directly to Definitions 6.3 and 6.5. Let

$$y = \lfloor 2^{n-1} sig(z) \rfloor = \lfloor 2^{n-1-e} z \rfloor = \lfloor x / 2^{e+1-n} \rfloor$$

and

$$f = 2^{n-1}sig(z) - y = 2^{n-1-e}z - y.$$

By Definition 2.2, $y = x[e : 1 + e - n]$. By Definition 6.1, $RTZ(z, n) = 2^{1+e-n}y$ and consequently $2^{1+e-n}f = z - RTZ(z, n)$ and $f = 2^{n-e-1}(z - RTZ(z, n))$. By Lemma 6.18,

$$\begin{aligned} z - RTZ(z, n) &= (RTZ(x, n+1) - RTZ(x, n)) + (z - RTZ(z, n+1)) \\ &= 2^{e-n}x[e-n] + (z - RTZ(z, n+1)). \end{aligned}$$

Thus,

$$f = 2^{n-e-1}(z - RTZ(z, n)) = \frac{1}{2}x[e-n] + 2^{n-e-1}(z - RTZ(z, n+1)).$$

By Lemma 6.5,

$$2^{n-e-1}(z - RTZ(z, n+1)) < 2^{n-e-1}2^{e-n} = \frac{1}{2}.$$

This leads to the following observations:

1. $f \geq \frac{1}{2}$ iff $x[e-n] = 1$.
2. $f > \frac{1}{2}$ iff $x[e-n] = 1$ and $z - RTZ(z, n+1) > 0$, but by Lemma 6.18,

$$\begin{aligned} z - RTZ(z, n+1) &= (z - x) + (x - RTZ(z, n+1)) \\ &= (z - x) + (RTZ(x, e+1) - RTZ(x, n+1)) \\ &= (z - x) + x[e-1-n : 0], \end{aligned}$$

and hence $f > \frac{1}{2}$ iff $x[e-n] = 1$ and either $x[e-1-n : 0] \neq 0$ or $z \neq x$.

Referring to Definition 6.5, we see that (1) is sufficient to complete the proof for the case $\mathcal{R} = RNA$.

For the case $\mathcal{R} = RNE$, it is clear from Definition 6.3 and (1) and (2) above that we need only show that y is even iff $x[1 + e - n] = 0$. But this is a consequence of the above equation $z = x[e : 1 + e - n]$. □

The final result of this section is a variation of Lemma 6.104 that allows us to compute the absolute value of a rounded result when the unrounded value z is negative, given a signed integer encoding x of $\lfloor z \rfloor$, i.e., $x = \lfloor z \rfloor + 2^k$, where $z \geq -2^k$.

Lemma 6.105 *Let $\mathcal{R}$ be a common rounding mode, $z \in \mathbb{R}$, $n \in \mathbb{N}$, and $k \in \mathbb{N}$. Assume that $0 < n < k$ and $-2^k \leq z < -2^n$. Let $x = \lfloor z \rfloor + 2^k$, $\tilde{x} = 2^k - x - 1$, and $e = expo(\tilde{x})$.*

(a) *If* $expo(z) \neq e$*, then* $z = -2^{e+1}$.
(b) z *is* n*-exact iff* $x[e-n:0] = 0$ *and* $z \in \mathbb{Z}$.
(c) *If any of the following conditions holds, then* $|\mathcal{R}(z,n)| = fp^+(RTZ(\tilde{x},n),n)$*, and otherwise* $|\mathcal{R}(z,n)| = RTZ(\tilde{x},n)$*:*

- $\mathcal{R} = RDN$ *or* $\mathcal{R} = RAZ$*;*
- $\mathcal{R} = RUP$ *or* $\mathcal{R} = RTZ$ *and both of the following are true:*
    - $x[e-n:0] = 0$*;*
    - $z \in \mathbb{Z}$*;*
- $\mathcal{R} = RNA$ *and at least one of the following is true:*
    - $x[e-n] = 0$*;*
    - $x[e-n-1:0] = 0$ *and* $z \in \mathbb{Z}$*;*
- $\mathcal{R} = RNE$ *and at least one of the following is true:*
    - $x[e-n] = 0$
    - $x[e+1-n] = x[e-n-1:0] = 0$ *and* $z \in \mathbb{Z}$.

Proof Let $f = z - \lfloor z \rfloor$. Then $x = (z-f) + 2^k$,

$$\tilde{x} = 2^k - x - 1 = 2^k - (z - f + 2^k) - 1 = -z - (1-f) = |z| - (1-f),$$

and $|z| = \tilde{x} + (1-f)$.

To prove (a), note that if $expo(z) \neq e$, then since $\tilde{x} \leq 2^{e+1} - 1$, we must have $\tilde{x} = 2^{e+1} - 1$, $f = 0$, and $|z| = 2^{e+1}$.

For the proof of (b) and (c), by Lemmas 2.18 and 6.16, we have

$$\begin{aligned}
|z| &= \tilde{x} + (1-f) \\
&= 2^{e+1-n}\tilde{x}[e:e+1-n] + \tilde{x}[e-n:0] + (1-f) \\
&= RTZ(\tilde{x},n) + \tilde{x}[e-n:0] + (1-f).
\end{aligned}$$

Suppose first that $f = x[e-n:0] = 0$. Then

$$|z| = RTZ(\tilde{x},n) + (2^{e+1-n} - 1) + 1 = fp^+(RTZ(\tilde{x},n),n) \in \mathbb{Z}$$

and the lemma claims that (b) z is n-exact and (c) $|\mathcal{R}(z,n)| = fp^+(RTZ(\tilde{x},n),n)$ for all $\mathcal{R}$. Both claims follow trivially from Lemmas 4.19, 6.7, and 6.90.

In the remaining case, $RTZ(\tilde{x},n) < |z| < fp^+(RTZ(\tilde{x},n),n)$. By Lemma 4.20, z is not n-exact, and (b) follows. The claim (c) will be derived from Lemma 6.104. Note that if $\hat{\mathcal{R}}$ is defined as in Lemma 6.87, then

$$|\mathcal{R}(z,n)| = -\mathcal{R}(z,n) = \hat{\mathcal{R}}(-z,n) = \hat{\mathcal{R}}(|z|,n),$$

and (c) may be restated as follows:

(c′) *If any of the following conditions holds, then* $\hat{\mathcal{R}}(|z|, n) = fp^+(RTZ(\tilde{x}, n), n)$, *and otherwise* $\hat{\mathcal{R}}(|z|, n) = RTZ(\tilde{x}, n)$:

- $\hat{\mathcal{R}} = RUP$ or $\hat{\mathcal{R}} = RAZ$;
- $\hat{\mathcal{R}} = RDN$ or $\hat{\mathcal{R}} = RTZ$ *and both of the following are true*:
  - $x[e - n : 0] = 0$;
  - $z \in \mathbb{Z}$;
- $\hat{\mathcal{R}} = RNA$ *and at least one of the following is true*:
  - $x[e - n] = 0$;
  - $x[e - n{-}1 : 0] = 0$ *and* $z \in \mathbb{Z}$;
- $\hat{\mathcal{R}} = RNE$ *and at least one of the following is true*:
  - $x[e - n] = 0$
  - $x[e + 1 - n] = x[e{-}n - 1 : 0] = 0$ *and* $z \notin \mathbb{Z}$.

We invoke Lemma 6.104 with $|z|$ and $\hat{\mathcal{R}}$ substituted for z and $\mathcal{R}$, respectively. This yields the following:

(c″) *If any of the following holds, then* $\hat{\mathcal{R}}(|z|, n) = fp^+(RTZ(\lfloor |z| \rfloor, n), n)$, *and otherwise* $\hat{\mathcal{R}}(|z|, n) = RTZ(\lfloor |z| \rfloor, n)$:

- $\hat{\mathcal{R}} = RUP$ or $\hat{\mathcal{R}} = RAZ$ *and at least one of the following is true*:
  - $\lfloor |z| \rfloor[e - n : 0] \neq 0$;
  - $z \notin \mathbb{Z}$.
- $\hat{\mathcal{R}} = RNA$ and $\lfloor |z| \rfloor[e - n] = 1$
- $\hat{\mathcal{R}} = RNE$, $\lfloor |z| \rfloor[e - n] = 1$, *and at least one of the following is true*:
  - $\lfloor |z| \rfloor[e - n{-}1 : 0] \neq 0$;
  - $z \notin \mathbb{Z}$;
  - $\lfloor |z| \rfloor[e + 1 - n] = 1$;

We must show that (c'') implies (c').

Suppose $f > 0$. Then $z \notin \mathbb{Z}$ and by Lemma 1.4,

$$\tilde{x} = 2^k - x - 1 = -\lfloor z \rfloor - 1 = \lfloor |z| \rfloor$$

and we may replace $\tilde{x}$ with $\lfloor |z| \rfloor$ in (c'). Furthermore, $|z|[e - n] = \tilde{x}[e - n]$, which implies $|z|[e - n] \neq x[e - n]$, and the claim (c') follows trivially from (c'').

In the final case, $f = 0$ and $x[e - n : 0] \neq 0$. Thus, $|z| = \tilde{x} + 1$ and $\tilde{x}[e - n : 0] < 2^{e+1-n} - 1$. Since

$$\begin{aligned}
|z| &= \tilde{x} + 1 \\
&= 2^{e+1-n}\tilde{x}[e : e + 1 - n] + \tilde{x}[e - n : 0] + 1 \\
&\leq 2^{e+1-n}(2^n - 1) + (2^{e+1-n} - 2) + 1 \\
&= 2^{e+1} - 1,
\end{aligned}$$

$expo(z) = e$. Since

$$|z|[e : e + 1 - n] = \left\lfloor \frac{|z|}{2^{e+1-n}} \right\rfloor = \tilde{x}[e : e + 1 - n],$$

$|z|[e + 1 - n] = \tilde{x}[e + 1 - n]$, which implies $|z|[e + 1 - n] \neq x[e + 1 - n]$. Lemma 6.16 implies $RTZ(|z|, n) = RTZ(\tilde{x}, n)$, and again we may replace $\tilde{x}$ with $\lfloor |z| \rfloor$ in (c'). Furthermore,

$$|z|[e - n : 0] = |z| \bmod 2^{e+1-n} = \tilde{x}[e - n : 0] + 1 = 2^{e+1-n} - x[e - n : 0].$$

If $x[e - n-1 : 0] = 0$, then $x[e - n] = 1$and

$$|z|[e - n : 0] = 2^{e+1-n} - x[e - n : 0] = 2^{e+1-n} - 2^{e-n} = 2^{e-n},$$

which implies $|z|[e - n] = 1$ and $|z|[e - n-1 : 0] = 0$. On the other hand, if $x[e - n-1 : 0] \neq 0$, then $\tilde{x}[e - n-1 : 0] < 2^{e-n} - 1$, which implies

$$|z|[e - n] = \tilde{x}[e - n] \neq x[e - n]$$

and

$$|z|[e - n-1 : 0] = \tilde{x}[e - n-1] + 1 \neq 0.$$

In either case, (c') again follows easily from (c''). □

6.6 Denormal Rounding

As we saw in Sect. 5.3, in order for a number x to be representable as a denormal in a format F, it must be $(prec(F) + expo(x) - expo(spn(F)))$-exact. This suggests the following definition of denormal rounding. Note that its arguments include the format itself, both parameters of which are required to determine the precision of the result.

Definition 6.10 Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$.

$$drnd(x, \mathcal{R}, F) = \mathcal{R}(x, prec(F) + expo(x) - expo(spn(F)).$$

While the conciseness of this formula is appealing, its computation for small x may involve negative-precision rounding, which has been observed to produce unintuitive results. In particular,

$$prec(F) + expo(x) - expo(spn(F)) \leq 0$$

$$\Leftrightarrow expo(x) \leq expo(spn(F)) - prec(F) = (1 - bias(F)) - prec(F)$$

$$\Leftrightarrow expo(x) < 2 - bias(F) - prec(F) = expo(spd(F))$$

$$\Leftrightarrow |x| < |spd(F)|.$$

We shall find, however (Lemma 6.108 below), that the results of rounding such tiny values are not unexpected.

Naturally, denormal rounding inherits many of the properties of normal rounding. It is not true in general that $sgn(drnd(x, \mathcal{R}, F)) = sgn(x)$ because a nonzero denormal may be rounded to 0, but we do have the following analog of Lemma 6.87.

Lemma 6.106 *Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$. Let $\mathcal{R}$ be a common rounding mode and let*

$$\hat{\mathcal{R}} = \begin{cases} RDN \text{ if } \mathcal{R} = RUP \\ RUP \text{ if } \mathcal{R} = RDN \\ \mathcal{R} \quad \text{otherwise.} \end{cases}$$

Then

$$drnd(-x, \mathcal{R}, F) = -drnd(x, \hat{\mathcal{R}}, F).$$

Proof This follows from Lemmas 6.87 and 4.5. □

Definition 6.10 admits the following alternative formulation:

Lemma 6.107 *Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$. Let $\mathcal{R}$ be a common rounding mode. Then*

$$drnd(x, \mathcal{R}, F) = \mathcal{R}(x + sgn(x)spn(F), prec(F)) - sgn(x)spn(F).$$

Proof Let $p = prec(F)$. We first consider the case $x \geq 0$ and apply Lemma 6.97, substituting $spn(F)$ for x, x for y, and $p + expo(x) - expo(spn(F))$ for k. Thus,

$$k' = k + expo(spn(F)) - expo(x) = p > 1,$$

and $spn(F)$ is $(k' - 1)$-exact by Lemma 4.10. Since

$$2^{expo(spn(F))} = spn(F) \leq spn(F) + x < 2 \cdot spn(F) = 2^{expo(spn(F))+1},$$

$expo(spn(F) + x) = expo(spn(F))$ and therefore

$$k'' = k + expo(spn(F) + x) - expo(x) = p$$

as well. Thus, we have

$$spn(F) + \mathcal{R}(x, k) = \mathcal{R}(spn(F) + x, k'')$$

and

$$\begin{aligned} drnd(x, \mathcal{R}, F) &= \mathcal{R}(x, p + expo(x) - expo(spn(F))) \\ &= \mathcal{R}(x, k) \\ &= \mathcal{R}(spn(F) + x, k'') - spn(F) \\ &= \mathcal{R}(spn(F) + x, n) - spn(F). \end{aligned}$$

The result may be extended to $x < 0$ by invoking Lemmas 6.87 and 6.106: if $\hat{\mathcal{R}}$ is defined as in these lemmas, then

$$\begin{aligned} drnd(x, \mathcal{R}, p, q) &= -drnd(-x, \hat{\mathcal{R}}, p, q) \\ &= -(\hat{\mathcal{R}}(-x + sgn(-x)spn(F), p) - sgn(-x)spn(F)) \\ &= -\hat{\mathcal{R}}(-x + sgn(-x)spn(F), p) + sgn(-x)spn(F) \\ &= \mathcal{R}(x + sgn(x)spn(F), p) - sgn(x)spn(F). \end{aligned}$$

□

We may now characterize the rounding of a denormal smaller than $spd(F)$:

Lemma 6.108 *Let F be a format, $x \in \mathbb{R}$, and $\mathcal{R}$ a common rounding mode.*

(a) If $0 < x < \frac{1}{2}spd(F)$, then

$$drnd(x, \mathcal{R}, F) = \begin{cases} spd(F) & \text{if } \mathcal{R} \in \{RAZ, RUP\} \\ 0 & \text{otherwise;} \end{cases}$$

(b) If $x = \frac{1}{2}spd(F)$, then

$$drnd(x, \mathcal{R}, F) = \begin{cases} spd(F) & \text{if } \mathcal{R} \in \{RAZ, RNA, RUP\} \\ 0 & \text{otherwise;} \end{cases}$$

(c) If $\frac{1}{2}spd(F) < x < spd(F)$, then

$$drnd(x, \mathcal{R}, F) = \begin{cases} 0 & \text{if } \mathcal{R} \in \{RTZ, RDN\} \\ spd(F) & \text{otherwise.} \end{cases}$$

Proof Let $p = prec(F)$, $q = expw(F)$, $a = spn(F) = 2^{2-2^{q-1}}$ and

$$b = fp^+(a, p) = a + 2^{expo(spn(F))+1-p} = a + 2^{3-2^{q-1}-p} = a + spd(F).$$

By Lemma 6.107,

$$drnd(x, \mathcal{R}, F) = \mathcal{R}(a + x, p) - a.$$

Case 1: $\mathcal{R} = RAZ$ or $\mathcal{R} = RUP$

By Lemma 6.21,

$$\mathcal{R}(a + x, p) = RAZ(a + x, p) \geq a + x > a,$$

and hence, by Lemmas 6.26 and 4.20,

$$RAZ(a + x, p) \geq b.$$

On the other hand, since

$$b = a + spd(F) > a + x,$$

Lemma 6.28 implies $b \geq RAZ(a + x, p)$, and therefore $RAZ(a + x, p) = b$, and

$$drnd(x, \mathcal{R}, F) = \mathcal{R}(a + x, p) - a = b - a = spd(F).$$

Case 2: $\mathcal{R} = RTZ$ or $\mathcal{R} = RDN$

First note that by Lemma 4.24,

$$fp^{-}(b, p) = fp^{-}(fp^{+}(a, p), p) = a.$$

Now by Lemma 6.3,

$$\mathcal{R}(a + x, p) = RTZ(a + x, p) \leq a + x < b,$$

and hence, by Lemmas 6.7 and 4.25,

$$RAZ(a + x, p) \leq a.$$

On the other hand, Lemma 6.9 implies $a \leq RTZ(a + x, p)$, and therefore $RTZ(a + x, p) = a$ and

$$drnd(x, \mathcal{R}, F) = \mathcal{R}(a + x, p) - a = a - a = 0.$$

Case 3: $\mathcal{R} = RNE$ or $\mathcal{R} = RNA$

$\mathcal{R}(a + x, p)$ is either a or b, and hence $drnd(x, \mathcal{R}, F)$ is either 0 or $spn(F)$, respectively.

Since

$$|(a + x) - RAZ(a + x, p)| = b - (a + x) = spd(F) - x$$

and

$$|(a + x) - RTZ(a + x, p)| = |(a + x) - a| = x,$$

the claims (a) and (c) follow from Lemmas 6.45 and 6.63. For the proof of (c), suppose $x = \frac{1}{2}spd(F)$. Then

$$a + x = 2^{2-2^{q-1}} + 2^{2-2^{q-1}-p} = 2^{2-2^{q-1}}(1 + 2^{-p})$$

and $sig(a + x) = 1 + 2^{-p}$. Thus, $a + x$ is $(p + 1)$-exact but not p-exact. The case $\mathcal{R} = RNA$ now follows from Lemma 6.54, and since b is not $(p - 1)$-exact, the case $\mathcal{R} = RNE$ follows from Lemma 6.69. □

As a consequence of Lemma 6.108 (a), for any given rounding mode, two sufficiently small numbers produce the same rounded result.

Corollary 6.109 *Let $x \in \mathbb{R}$, $y \in \mathbb{R}$. Let F be a format and $\mathcal{R}$ a common rounding mode. If*

$$0 < x < \frac{1}{2}spd(F)$$

and

$$0 < y < \frac{1}{2}spd(F),$$

then

$$drnd(x, \mathcal{R}, F) = drnd(y, \mathcal{R}, F).$$

A denormal is always rounded to a representable number, which may be denormal, 0, or the smallest representable normal.

Lemma 6.110 *Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$. Let $\mathcal{R}$ be a common rounding mode. Then one of the following is true:*

(a) $drnd(x, \mathcal{R}, F) = 0$;
(b) $drnd(x, \mathcal{R}, F) = sgn(x)spn(F)$;
(c) $drnd(x, \mathcal{R}, F)$ is representable as a denormal in F.

Proof By Lemmas 6.106 and 6.108, we may assume $x \geq spd(F)$. Let $p = prec(F)$. Then

$$expo(x) \geq expo(spn(F)) = 2 - bias(F) - p.$$

Since $x < spn(F) = 2^{1-bias(F)}$, $expo(x) \leq -bias(F)$, and

$$2 - p \leq expo(x) + bias(F) \leq 0.$$

Let $d = drnd(x, \mathcal{R}, F)$. By Lemma 6.89, d is $(p + expo(x) - expo(spn(F)))$-exact. If $expo(d) = expo(x)$, then d is representable as a denormal. If not, then

Lemma 6.94 implies $d = 2^{expo(x)+1}$. In this case, either $expo(x) = -bias(F)$ and

$$d = 2^{1-bias(F)} = spn(F),$$

or $expo(x) + bias(F) < 0$ and

$$expo(d) + bias(F) = 1 + expo(x) + bias(F) \leq 0,$$

which implies that d is representable as a denormal. □

Lemma 6.111 *If x is representable as a denormal in F and $\mathcal{R}$ is a common rounding mode, then*

$$drnd(x, \mathcal{R}, F) = x.$$

Proof Let $p = prec(F)$. By Definition 5.17, x is $(p + expo(x) - expo(spn(F)))$-exact. Therefore, by Lemma 6.90,

$$drnd(x, \mathcal{R}, F) = \mathcal{R}(x, p + expo(x) - expo(spn(F))) = x.$$

□

Lemma 6.112 *If F is a format, $\mathcal{R}$ is a common rounding mode, and $x \in \mathbb{R}$ with*

$$2 - bias(F) - prec(F) \leq expo(x) \leq -bias(F),$$

then $drnd(x, \mathcal{R}, F) = x$ iff x is $(expo(x) + bias(F) + prec(F) - 1)$-exact.

Proof Suppose $drnd(x, \mathcal{R}, F) = x$. Since $0 < |x| < spn(F)$, Lemma 6.110 guarantees that x is representable as a denormal, which implies that x is $(expo(x) + bias(F) + prec(F) - 1)$-exact.

On the other hand, if x is $(expo(x) + bias(F) + prec(F) - 1)$-exact, then x is representable by Definition 5.17 and $drnd(x, \mathcal{R}, F) = x$ by Lemma 6.111. □

Lemma 6.113 *Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$. Let a be representable as a denormal in F and let $\mathcal{R}$ be a common rounding mode.*

(a) If $a \geq x$, then $a \geq drnd(x, \mathcal{R}, F)$.
(b) If $a \leq x$, then $a \leq drnd(x, \mathcal{R}, F)$.

Proof By Lemma 6.108, we may assume $|x| \geq spd(F)$. By Definition 5.17, a is $(prec(F) + expo(x) - expo(spn(F)))$-exact. The claim follows from Lemma 6.91.

□

The defining characteristics of the directed rounding modes are inherited by denormal rounding.

Lemma 6.114 *Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$.*

(a) $|drnd(x, RTZ, F)| \leq |x|$.
(b) $|drnd(x, RAZ, F)| \geq |x|$.
(c) $drnd(x, RDN, F) \leq x$.
(d) $drnd(x, RUP, F) \geq x$.

Proof This is a consequence of Lemmas 6.108, 6.3, 6.21, and 6.86. □

Denormal rounding error is bounded by the distance between successive representable numbers (see Lemma 5.10).

Lemma 6.115 *Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$. Let $\mathcal{R}$ be a common rounding mode. Then*

$$|x - drnd(x, \mathcal{R}, F)| < spd(F).$$

Proof Again we may assume $|x| \geq spd(F)$. Let $p = prec(F)$. By Lemma 6.93,

$$\begin{aligned} |x - drnd(x, \mathcal{R}, F)| &= |x - \mathcal{R}(x, p + expo(x) - expo(spn(F)))| \\ &< 2^{expo(x)+1-(p+expo(x)-expo(spn(F)))} \\ &= 2^{expo(spn(F))-(p-1)} \\ &= spd(F). \end{aligned}$$

□

Naturally, denormal rounding to nearest returns the representable number that is closest to its argument.

Lemma 6.116 *Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$. Let a be representable as a denormal in F. Then*

$$|x - drnd(x, RNE, F)| \leq |x - a|$$

and

$$|x - drnd(x, RNA, F)| \leq |x - a|.$$

Proof By Definition 5.17, a is $(prec(F) + expo(x) - expo(spn(F)))$-exact. The result follows from Lemmas 6.47 and 6.66. □

The next lemma, which pertains to the detection of floating-point underflow, warrants some motivation. Let x be the precise numerical result of an arithmetic operation, to be rounded according to a mode $\mathcal{R}$ and encoded in a format F with precision p. Most implementations first compute the value $r = rnd(x, \mathcal{R}, p)$, using an internal format with a sufficiently wide exponent field to accommodate this result. According to the x86 architectural definition, underflow occurs when $|r| < spn(F)$.

If this occurs and the underflow mask is set, then the value $d = dnrd(x, \mathcal{R}, F)$ is computed and returned, and the underflow flag is set iff $d \neq x$.

There are, however, implementations that compute d directly in the event that $|u| < spn(F)$, without computing r. The requirement of correctly setting the underflow flag then presents a problem, since r may lie below the normal range when d does not. Thus, for such an implementation, if $|x| < |d| = spn(F)$, then extra logic is required to determine whether $|r| < spn(F)$. On the other hand, there is no such ambiguity requiring extra logic for the case $|d| < spn(F)$, since the following lemma guarantees that $|r| < spn(F)$ as well.

Lemma 6.117 *Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$. Let $\mathcal{R}$ be a common rounding mode. If*

$$|rnd(x, \mathcal{R}, prec(F))| = spn(F),$$

then

$$|drnd(x, \mathcal{R}, F)| = spn(F).$$

Proof Note that the hypothesis implies $|x| \geq spd(F)$. Let $p = prec(F)$. By Lemma 6.94, $expo(x) = expo(spn(F)) - 1$, and hence

$$p + expo(x) - expo(spn(F) = p - 1.$$

Since $drnd(x, \mathcal{R}, F) = rnd(x, \mathcal{R}, p + expo(x) - expo(spn(F)))$, the claim follows from Lemma 6.102 with $m = p - 1$ and $n = p$. □

The final result of this section pertains to the setting of the precision flag in the event of underflow and is relevant to the formal architectural specifications discussed in Part IV: if d is exact, then so is r:

Lemma 6.118 *Let F be a format and let $x \in \mathbb{R}$, $|x| < spn(F)$. Let $\mathcal{R}$ be a common rounding mode. If $drnd(x, \mathcal{R}, F) = x$, then $rnd(x, \mathcal{R}, prec(F)) = x$.*

Proof Let $p = prec(F)$. By Lemma 6.110, x is representable as a denormal in F, which implies that x is $(expo(x) + bias(F) + p - 1)$-exact. Since $expo(x) < expo(spn(F)) = 1 - bias(F)$, $expo(x) + bias(F) + p - 1 < p$. By Lemma 4.9, x is p-exact, and the claim follows from Lemma 6.90. □

Chapter 7
IEEE-Compliant Square Root

Many of the preceding results are propositions pertaining to real variables, which are formalized by ACL2 events in which these variables are restricted to the rational domain. Many of the lemmas of this chapter similarly apply to arbitrary real numbers, but in light of our present focus, these results are formulated to correspond more closely with their formal versions. Apart from the informal discussion immediately below, the lemmas themselves contain no references to the real numbers or the square root function.

Establishing IEEE compliance of a floating-point square root module entails proving that the final value r computed for a given radicand x, rounding mode $\mathcal{R}$, and precision n satisfies

$$r = \mathcal{R}(\sqrt{x}, n). \tag{7.1}$$

We would like to formulate a proposition of rational arithmetic that is transparently equivalent to (7.1). This requirement is satisfied by the following criterion:

For all positive rational numbers ℓ *and* h, *if* $\ell^2 \leq x \leq h^2$, *then*

$$\mathcal{R}(\ell, n) \leq r \leq \mathcal{R}(h, n). \tag{7.2}$$

Obviously, the monotonicity of rounding (Lemma 6.95) and of the square root ensure that (7.1) implies (7.2). On the other hand, suppose that (7.2) holds. According to Lemma 6.100, either $\sqrt{x}$ is $(n+1)$-exact (and, in particular, rational) or for some $\epsilon > 0$, $\mathcal{R}(y, n) = \mathcal{R}(\sqrt{x}, n)$ for all y satisfying $|y - \sqrt{x}| < \epsilon$. In either case, there exist $\ell \in \mathbb{Q}$ and $h \in \mathbb{Q}$ such that $\ell \leq \sqrt{x} \leq h$ and $\mathcal{R}(\ell, n) = \mathcal{R}(\sqrt{x}, n) = \mathcal{R}(h, n)$. Since $\ell^2 \leq x \leq h^2$,

$$\mathcal{R}(\sqrt{x}, n) = \mathcal{R}(\ell, n) \leq r \leq \mathcal{R}(h, n) = \mathcal{R}(\sqrt{x}, n)$$

and hence $r = \mathcal{R}(\sqrt{x}, n)$.

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_7

Thus, we would like to prove formally that (7.2) is satisfied by the value r computed by a square root module of interest. For this purpose, it will be useful to have a function that computes, for given x and n, a rational number q that satisfies

$$\mathcal{R}(q, n) = \mathcal{R}(\sqrt{x}, n). \tag{7.3}$$

We shall define a conceptually simple (albeit computationally horrendous) rational function $\sqrt[(k)]{x}$ that serves this need. The definition is motivated by Lemma 6.101, which guarantees that if we are able to arrange that

$$\sqrt[(k)]{x} = RTO(\sqrt{x}, k), \tag{7.4}$$

where $k \geq n + 2$, then (7.3) holds for $q = \sqrt[(k)]{x}$. Of course, (7.4) will not be our formal definition of $\sqrt[(k)]{x}$, nor shall we prove any instance of (7.3). However, after formulating the definition, we shall prove the following (Lemma 7.17):

For all positive rationals ℓ and h and positive integers k and n, if $\ell^2 \leq x \leq h^2$ and $k \geq n + 2$, then

$$\mathcal{R}(\ell, n) \leq \mathcal{R}(\sqrt[(k)]{x}, n) \leq \mathcal{R}(h, n). \tag{7.5}$$

Thus, in order to prove that a computed value r satisfies (7.2), it will suffice to show that $r = \mathcal{R}(\sqrt[(k)]{x}, n)$ for some $k \geq n + 2$. This is the strategy followed in the correctness proof of Chap. 19.

7.1 Truncated Square Root

The first step toward the definition of $\sqrt[(n)]{x}$ is the following recursive function, the name of which is motivated by the unproven observation that for $\frac{1}{4} \leq x < 1$,

$$\textit{rtz-sqrt}(x, n) = RTZ(\sqrt{x}, n).$$

Definition 7.1 Let $x \in \mathbb{R}$ and $n \in \mathbb{N}$. If $n = 0$, then $\textit{rtz-sqrt}(x, n) = 0$ and if $n > 0$ and $z = \textit{rtz-sqrt}(x, n - 1)$, then

$$\textit{rtz-sqrt}(x, n) = \begin{cases} z & \text{if } (z + 2^{-n})^2 > x \\ z + 2^{-n} & \text{if } (z + 2^{-n})^2 \leq x. \end{cases}$$

Lemma 7.1 *Let $x \in \mathbb{Q}$ and $n \in \mathbb{N}$. If $x \geq \frac{1}{4}$, then*

$$\frac{1}{2} \leq \textit{rtz-sqrt}(x, n) \leq 1 - 2^{-n}.$$

Proof If $n = 1$, then $\textit{rtz-sqrt}(x, n) = \frac{1}{2}$ and the claim is trivial. Proceeding by induction, let $n > 1$, $z = \textit{rtz-sqrt}(x, n-1)$, and $w = rtz(x, n)$, and assume that $\frac{1}{2} \le z \le 1 - 2^{1-n}$. If $w = z$, the claim follows trivially; otherwise, $w = z + 2^{-n}$ and

$$\frac{1}{2} \le z < w = z + 2^{-n} \le (1 - 2^{1-n}) + 2^{-n} = 1 - 2^{-n}.$$

□

Corollary 7.2 *Let $x \in \mathbb{Q}$ $n \in \mathbb{Z}^+$. If $x \ge \frac{1}{4}$, then $expo(\textit{rtz-sqrt}(x, n)) = -1$.*

Lemma 7.3 *Let $x \in \mathbb{Q}$ and $n \in \mathbb{Z}^+$. If $x \ge \frac{1}{4}$, then $\textit{rtz-sqrt}(x, n)$ is n-exact.*

Proof The claim is trivial for $n = 0$. Let $n > 1$, $z = \textit{rtz-sqrt}(x, n-1)$, and $w = rtz(x, n)$, and assume that z is $(n-1)$-exact, i.e., $2^{n-1}z \in \mathbb{Z}$. Then either $w = z$ and $2^n w = 2(2^{n-1}z) \in \mathbb{Z}$ or

$$2^n w = 2^n(z + 2^{-n}) = 2(2^{n-1}z) + 1 \in \mathbb{Z}.$$

□

Lemma 7.4 *Let $x \in \mathbb{Q}$ and $n \in \mathbb{N}$. Assume that $\frac{1}{4} \le x < 1$ and let $w = \textit{rtz-sqrt}(x, n)$. Then $w^2 \le x < (w + 2^{-n})^2$.*

Proof The claim is trivial for $n = 0$. Let $n > 0$, $z = \textit{rtz-sqrt}(x, n-1)$, and assume that $z^2 \le x < (z + 2^{1-n})^2$. If $x < (z + 2^{-n})^2$ and $w = z$, the claim is trivial. Otherwise, $x \ge (z + 2^{-n})$, $w = z + 2^{-n}$, and

$$w^2 = (z + 2^{-n})^2 \le x \le (z + 2^{1-n})^2 = (w + 2^{-n})^2.$$

□

According to the next lemma, $\textit{rtz-sqrt}(x, n)$ is uniquely determined by the above properties.

Lemma 7.5 *Let $x \in \mathbb{Q}$, $a \in \mathbb{Q}$, and $n \in \mathbb{Z}^+$. Assume that $\frac{1}{4} \le x < 1$ and $a \ge \frac{1}{2}$. If a is n-exact and $a^2 \le x < (a + 2^{-n})^2$, then $a = \textit{rtz-sqrt}(x, n)$.*

Proof Let $w = \textit{rtz-sqrt}(x, n)$. If $a < w$, then by Lemma 4.20,

$$w \ge \textit{fp}^+(a, n) = a + 2^{expo(a)+1-n} \ge a + 2^{-n},$$

which implies $w^2 \ge (a + 2^{-n})^2 > x$, contradicting Lemma 7.4. But if $a > w$, then

$$a \ge \textit{fp}^+(w, n) = w + 2^{-n},$$

and by Lemma 7.4, $a^2 \ge (w + 2^{-n})^2 > x$, contradicting our hypothesis. □

We have the following variation of Lemma 6.12.

Lemma 7.6 *Let $x \in \mathbb{Q}$, $m \in \mathbb{Z}^+$, and $n \in \mathbb{Z}^+$. If $x \geq \frac{1}{4}$ and $n \geq m$, then*

$$RTZ(rtz\text{-}sqrt(x, n), m) = rtz\text{-}sqrt(x, m).$$

Proof The case $m = n$ follows from Lemmas 6.8 and 7.3. We proceed by induction on $n - m$. Let $1 < m \leq n$ and assume that $RTZ(rtz\text{-}sqrt(x, n), m) = rtz\text{-}sqrt(x, m)$. Then by Lemma 6.12,

$$\begin{aligned} RTZ(rtz\text{-}sqrt(x, n), m - 1) &= RTZ(RTZ(rtz\text{-}sqrt(x, n), m), m - 1) \\ &= RTZ(rtz\text{-}sqrt(x, m), m - 1), \end{aligned}$$

and we need only show that $RTZ(rtz\text{-}sqrt(x, m), m - 1) = rtz\text{-}sqrt(x, m - 1)$. Let $w = rtz\text{-}sqrt(x, m)$ and $z = rtz\text{-}sqrt(x, m - 1)$. If $w = z$, then w is $(n - 1)$-exact by Lemma 7.3 and $RTZ(w, n - 1) = w = z$ by Lemma 6.8. But otherwise, $w = z + 2^{-n}$, $2^{n-1}z \in \mathbb{Z}$ by Corollary 7.2, and hence, by Definition 6.1,

$$\begin{aligned} RTZ(w, n - 1) &= 2^{1-n}\lfloor 2^{n-1}w \rfloor \\ &= 2^{1-n}\lfloor 2^{n-1}(z + 2^{-n}) \rfloor \\ &= 2^{1-n}\lfloor 2^{n-1}z + 1 \rfloor \\ &= 2^{1-n}(2^{n-1}z) \\ &= z. \end{aligned}$$

□

7.2 Odd-Rounded Square Root

The name of the following function is motivated by the (once again unproven) observation that for $\frac{1}{4} \leq x < 1$,

$$rto\text{-}sqrt(x, n) = RTO(\sqrt{x}, n).$$

Definition 7.2 Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}^+$, and let $z = rtz\text{-}sqrt(x, n - 1)$. Then

$$rto\text{-}sqrt(x, n) = \begin{cases} z & \text{if } x \leq z^2 \\ z + 2^{-n} & \text{if } x > z^2. \end{cases}$$

Lemma 7.7 *Let $x \in \mathbb{Q}$ and $n \in \mathbb{Z}^+$. If $x \geq \frac{1}{4}$, then*

$$\frac{1}{2} \leq rto\text{-}sqrt(x, n) \leq 1 - 2^{-n}.$$

Proof If $n = 1$, then $rto\text{-}sqrt(x, n) = \frac{1}{2}$ and the claim is trivial. Let $n > 1$ and $z = rtz\text{-}sqrt(x, n-1)$. By Lemma 7.1, $\frac{1}{2} \leq z < 1$, which implies

$$\frac{1}{2} \leq z \leq rto\text{-}sqrt(x, n) \leq z + 2^{-n} \leq (1 - 2^{1-n}) + 2^{-n} = 1 - 2^{-n}.$$

□

Corollary 7.8 *Let $x \in \mathbb{Q}$ $n \in \mathbb{Z}^+$. If $x \geq \frac{1}{4}$, then $expo(rto\text{-}sqrt(x, n)) = -1$.*

Lemma 7.9 *Let $x \in \mathbb{Q}$ and $n \in \mathbb{Z}^+$. If $x \geq \frac{1}{4}$, then $rto\text{-}sqrt(x, n)$ is n-exact.*

Proof Let $z = rtz\text{-}sqrt(x, n-1)$ and $w = rto\text{-}sqrt(x, n)$. By Corollaries 7.2 and 7.8, $expo(z) = expo(w) = -1$. By Lemma 7.3, $2^{n-1}z \in \mathbb{Z}$. Consequently, since w is either z or $z + 2^{-n}$, $2^n w \in \mathbb{Z}$, i.e., w is n-exact. □

Lemma 7.10 *Let $x \in \mathbb{Q}$, $m \in \mathbb{Z}^+$, and $n \in \mathbb{N}$. Assume that $\frac{1}{4} \leq x < 1$ and $2 \leq n \leq m$. Then*

$$rto(rto\text{-}sqrt(x, m), n) = rto\text{-}sqrt(x, n).$$

Proof We first consider the case $n = m - 1$. Let $z_1 = rtz\text{-}sqrt(x, m-2)$, $w_1 = rto\text{-}sqrt(x, m-1)$, $z_2 = rtz\text{-}sqrt(x, m-1)$, and $w_2 = rto\text{-}sqrt(x, m)$. We shall show that $rto(w_2, m-1) = w_1$. Note that by Lemmas 7.2, 7.6, and 7.4, $\frac{1}{2} \leq z_1^2 \leq z_2^2 \leq x$.

Case 1: $z_1 = z_2$ and $z_2^2 < x$.

$z_1 = w_1 = z_2 = w_2$. Since w_2 is $(m-1)$-exact, Lemma 6.74 implies $rto(w_2, m-1) = w_2 = w_1$.

Case 2: $z_1 = z_2$ and $z_2^2 < x$.

Since w_1 is $(m-2)$-exact, Lemma 4.20 implies that $w_1 = z_1 + 2^{1-n}$ is not $(m-2)$-exact; similarly, since w_2 is $(m-1)$-exact, $w_2 = z_2 + 2^{-m}$ is not $(m-1)$-exact. Therefore,

$$rto(w_2, m-1) = RTZ(w_2, m-2) + 2^{1-n} = z_1 + 2^{1-m} = w_1.$$

Case 3: $z_1 < z_2$ and $z_2^2 = x$.

By Lemma 7.3, z_1 is $(m-1)$-exact and z_2 is $(m-2)$-exact. By Lemma 7.6, $z_1 = RTZ(z_2, m-2) < z_2$, and it follows from Lemma 6.9 that $z_2 = z_1 + 2^{1-m}$. Thus, $w_1 = z_2 = w_2$ and by Lemma 6.74, $rto(w_2, m-1) = w_2 = w_1$.

Case 4: $z_1 < z_2$ and $z_2^2 < x$.

In this case, $w_1 = z_2 = z_1 + 2^{1-m}$ and $w_2 = z_2 + 2^{-m} = z_1 + 2^{1-m} + 2^{-m}$, which is not $(m-2)$-exact. Thus,

$$rto(w_2, m-1) = RTZ(w_2, m-2) + 2^{1-m} = z_1 + 2^{1-m} = w_1.$$

The proof is completed by induction on m. If $m > n$, then by Lemma 6.81,

$$\begin{aligned} rto(rto\text{-}sqrt(x, m), n) &= rto(rto(rto\text{-}sqrt(x, m), m - 1), n) \\ &= rto(rto\text{-}sqrt(x, m - 1), n) \\ &= rto\text{-}sqrt(x, n). \ \Box \end{aligned}$$

Lemma 7.11 *Let $x \in \mathbb{Q}$, $\ell \in \mathbb{Q}$,$h \in \mathbb{Q}$, and $n \in \mathbb{Z}^+$. Assume that $\frac{1}{4} \leq x < 1$, $h > 0$, and $\ell^2 \leq x \leq h^2$. Then*

$$rto(\ell, n) \leq rto\text{-}sqrt(x, n) \leq rto(h, n).$$

Proof Let $z = rtz\text{-}sqrt(x, n - 1)$ and $w = rto\text{-}sqrt(x, n)$. Suppose $z^2 = x$. Then $w = z$, $\ell^2 \leq x = w^2$, and hence $\ell \leq w$. By Lemmas 6.79, 6.74, and 7.3,

$$rto(l, n) \leq rto(w, n) = w.$$

Thus, we may assume $z^2 < x$ and $w = z + 2^{-n}$. By Lemma 7.4, $\ell^2 \leq x < w^2$, and hence $\ell < w = fp^+(z, n - 1)$. It follows from Lemmas 6.3, 6.7, and 4.20 that $RTZ(\ell, n - 1) \leq z$. Therefore,

$$rto(\ell, n) \leq RTZ(\ell, n - 1) + 2^{1+expo(\ell)-n} \leq z + 2^{-n} = w.$$

To prove the second inequality, we note that if $h \geq w$, then by Lemmas 6.79, 6.74, and 7.3,

$$rto(h, n) \geq rto(w, n) = w.$$

Therefore, we may assume that $h < w$. If $z^2 = x$, then $w = z$, $h^2 \geq x = w^2$, and $h \geq w$. Thus, by Lemma 7.4, $z^2 < x$ and $w = z + 2^{-n} = fp^+(z, n - 1)$. Since $h^2 \geq x > z^2$, $h > z$. It follows from Lemma 6.9 that $RTZ(h, n - 1) \geq a$. By Lemma 4.20, h is not n-exact, and hence

$$rto(h, n) = RTZ(h, n - 1) + 2^{-n} \geq z + 2^{-n} = w. \ \Box$$

Lemma 7.12 *Let $x \in \mathbb{Q}$, $q \in \mathbb{Q}$, and $n \in \mathbb{Z}^+$. Assume that $\frac{1}{4} \leq x < 1$, $q > 0$, and q is $(n - 1)$-exact. Then*

(a) $q^2 < x \Leftrightarrow q < rto\text{-}sqrt(x, n)$;
(b) $q^2 > x \Leftrightarrow q > rto\text{-}sqrt(x, n)$.

Proof Let $z = rtz\text{-}sqrt(x, n - 1)$ and $w = rto\text{-}sqrt(x, n)$. If $q^2 > x$, then by Lemma 7.4, $q^2 > z^2$, so that $q > z$ and by Lemma 4.20,

$$q \geq z + 2^{1-n} > z + 2^{-n} \geq w.$$

We may assume, therefore, that $q^2 \leq x < (z + 2^{-n})^2$, and hence $q < z + 2^{-n}$. We must show that $q < x^2$ iff $q < w$. By Lemma 4.20, $q \leq z$. If $q < z$, then $q < x^2$ and $q < w$. If $q = z = x^2$, then $q = z = w$. Finally, if $q = z < x^2$, then $q = z < z + 2^{-n} = w$. □

7.3 IEEE-Rounded Square Root

The desired approximation function is a simple generalization of *rto-sqrt* to arbitrary positive rationals:

Definition 7.3 Let $x \in \mathbb{Q}$ and $n \in \mathbb{Z}^+$ with $x > 0$. Let $e = \left\lfloor \frac{expo(x)}{2} \right\rfloor + 1$. Then

$$\sqrt[(n)]{x} = 2^e rto\text{-}sqrt(2^{-2e}x, n).$$

Lemma 7.13 *Let $x \in \mathbb{Q}$, $x > 0$, $e = \left\lfloor \frac{expo(x)}{2} \right\rfloor + 1$, and $x' = 2^{-2e}x$. Then $\frac{1}{4} \leq x' < 1$.*

Proof Since

$$\frac{expo(x)}{2} - 1 < \left\lfloor \frac{expo(x)}{2} \right\rfloor \leq \frac{expo(x)}{2},$$

we have

$$expo(x) < 2\left\lfloor \frac{expo(x)}{2} \right\rfloor + 2 = 2e$$

and

$$expo(x) \geq 2\left\lfloor \frac{expo(x)}{2} \right\rfloor = 2e - 2.$$

By Lemma 4.6, $-2 \leq expo(x') < 0$ and the lemma follows. □

Lemma 7.14 *Let $x \in \mathbb{Q}$ and $n \in \mathbb{Z}^+$. If $\frac{1}{4} \leq x < 1$, then*

$$\sqrt[(n)]{x} = rto\text{-}sqrt(x, n).$$

Proof Since $expo(x) \in \{-2, -1\}$, $\left\lfloor \frac{expo(x)}{2} \right\rfloor = -1$ and $e = 0$. □

Lemma 7.15 *Let $x \in \mathbb{Q}$ and $n \in \mathbb{Z}^+$ with $x > 0$. For all $k \in \mathbb{Z}$,*

$$\sqrt[(n)]{2^{2k}x} = 2^k \sqrt[(n)]{x}.$$

Proof Let $x' = 2^{2k}x$, $e = \left\lfloor \frac{expo(x)}{2} \right\rfloor + 1$, and

$$e' = \left\lfloor \frac{expo(x')}{2} \right\rfloor + 1 = \left\lfloor \frac{expo(x)}{2} + k \right\rfloor + 1 = e + k.$$

Then

$$\begin{aligned}
\sqrt[(n)]{x'} &= 2^{e'} rto\text{-}sqrt(2^{-2e'}x', n) \\
&= 2^{e+k} rto\text{-}sqrt(2^{-2(e+k)}2^{2k}x, n) \\
&= 2^k \left(2^e rto\text{-}sqrt(2^{2e}x, n)\right) \\
&= 2^k \sqrt[(n)]{x}. \ \Box
\end{aligned}$$

Lemma 7.16 *Let $x \in \mathbb{Q}$, $k \in \mathbb{N}$, $m_1 \in \mathbb{N}$, and $n_2 \in \mathbb{N}$ with $x > 0$ and $2 < k + 2 \leq m \leq n$ and let $\mathcal{R}$ be a common rounding mode. Then*

$$\mathcal{R}(\sqrt[(m)]{x}, k) = \mathcal{R}(\sqrt[(n)]{x}, k).$$

Proof Let $e = \left\lfloor \frac{expo(x)}{2} \right\rfloor + 1$. By Definition 7.3 and Lemmas 7.11 and 6.80,

$$\begin{aligned}
\mathcal{R}(\sqrt[(m)]{x}, k) &= \mathcal{R}(2^e rto\text{-}sqrt(2^{-2e}x, m), k) \\
&= 2^e \mathcal{R}(rto\text{-}sqrt(2^{-2e}x, m), k) \\
&= 2^e \mathcal{R}(rto\text{-}sqrt(2^{-2e}x, n), k) \\
&= \mathcal{R}(2^e rto\text{-}sqrt(2^{-2e}x, n), k) \\
&= \mathcal{R}(\sqrt[(n)]{x}, k). \ \Box
\end{aligned}$$

The next lemma establishes the critical property of $\sqrt[(k)]{x}$ discussed at the beginning of this chapter.

Lemma 7.17 *Let $x \in \mathbb{Q}$, $\ell \in \mathbb{Q}$, $h \in \mathbb{Q}$ $n \in \mathbb{Z}^+$, and $k \in \mathbb{Z}^+$. Assume that $x > 0$ $h > 0$, $k \geq n + 2$, and $\ell^2 \leq x \leq h^2$. Let $\mathcal{R}$ be a common rounding mode. Then*

$$\mathcal{R}(\ell, n) \leq \mathcal{R}(\sqrt[(k)]{x}, n) \leq \mathcal{R}(h, n).$$

Proof Let $e = \left\lfloor \frac{expo(x)}{2} \right\rfloor + 1$, $x' = 2^{-2e}x$, $\ell' = 2^{-e}\ell$, and $h' = 2^{-e}h$. By Lemmas 7.13 and 7.11,

$$RTO(\ell', k) \leq rto\text{-}sqrt(x', k) \leq RTO(h', k),$$

or

$$RTO(2^{-k}\ell, k) \leq 2^{-k} \sqrt[(k)]{x} \leq RTO(2^{-k}h, k).$$

By Lemma 6.80,

$$RTO(\ell, k) \leq \sqrt[(k)]{x} \leq RTO(h, k),$$

and by Lemma 6.101,

$$\mathcal{R}(\ell, n) = \mathcal{R}(RTO(\ell, k), n) \leq \mathcal{R}(\sqrt[(k)]{x}, n) \leq \mathcal{R}(RTO(h, k), n) = \mathcal{R}(h, n). \ \Box$$

Our final lemma, which is also required for the proof of Chap. 19, warrants some motivation. In practice, a typical implementation of a subtractive square root algorithm produces a final truncated approximation q of the square root and a remainder that provides a comparison between q^2 and the radicand x. A final rounded result r is derived from this approximation in accordance with a given rounding mode $\mathcal{R}$ and precision n. In order to apply (7.5), we would like to show that $r = \mathcal{R}(\sqrt[(k)]{x}, n)$ for some appropriate k. This may be done, for example, by invoking Lemma 6.104 with q and $\sqrt[(k)]{x}$ substituted for x and z, respectively. But this requires showing that $q = RTZ(\sqrt[(k)]{x}, n)$ and determining whether $q = \sqrt[(k)]{x}$. Thus, we require a means of converting inequalities relating q^2 and x to inequalities relating q and $\sqrt[(k)]{x}$. This is achieved by the following:

Lemma 7.18 *Let $x \in \mathbb{Q}$, $q \in \mathbb{Q}$, and $n \in \mathbb{N}$. Assume that $x > 0$, $q > 0$, $n > 1$, and q is $(n-1)$-exact. Then*

(a) $q^2 < x \Leftrightarrow q < \sqrt[(n)]{x}$;
(b) $q^2 > x \Leftrightarrow q > \sqrt[(n)]{x}$;
(c) $q^2 = x \Leftrightarrow q = \sqrt[(n)]{x}$.

Proof Let $e = \left\lfloor \frac{expo(x)}{2} \right\rfloor + 1$, $x' = 2^{-2e}x$, and $q' = 2^{-e}q$. Then $\frac{1}{4} \leq x' < 1$ and $\sqrt[(n)]{x} = 2^e rto\text{-}sqrt(x', n)$. By Lemma 7.12,

$$\begin{aligned} q^2 < x &\Leftrightarrow q'^2 < x' \\ &\Leftrightarrow q' < rto\text{-}sqrt(x', n) \\ &\Leftrightarrow 2^{-e}q < 2^{-e} \sqrt[(n)]{x} \\ &\Leftrightarrow q < \sqrt[(n)]{x}. \end{aligned}$$

The proof of (b) is similar, and (c) follows. □

Corollary 7.19 *Let $x \in \mathbb{Q}$ and $n \in \mathbb{N}$ with $x > 0$ and $n > 1$. If $\sqrt[(n)]{x}$ is $(n-1)$-exact, then $(\sqrt[(n)]{x})^2 = x$.*

Proof Instantiate Lemma 7.18 with $q = \sqrt[(n)]{x}$. □

Corollary 7.20 *Let $x \in \mathbb{Q}$, $k \in \mathbb{N}$, $n \in \mathbb{N}$, and $m \in \mathbb{N}$ with $x > 0$, $k > 1$, $n > k$, and $m > k$. If $\sqrt[(n)]{x}$ is $(n-1)$-exact, then $\sqrt[(m)]{x} = \sqrt[(n)]{x}$.*

Proof By Corollary 7.19, $(\sqrt[(n)]{x})^2 = x$. The corollary again follows from Lemma 7.18. □

Lemma 7.18 is also critical in the detection of floating-point precision exceptions. As described more fully in Sects. 12.5, 13.5, and 14.3, this exception is signaled when an instruction returns a rounded result r that differs from the precise mathematical value u of an operation. But in the case of the square root, the ACL2 formalization compares r to $\sqrt[(p+2)]{x}$ rather than $u = \sqrt{x}$, where p is the target precision. This is justified by (c) above, from which it follows that $r = \sqrt{x}$ iff $r = \sqrt[(p+2)]{x}$.

Part III
Implementation of Elementary Operations

The marvel of efficient and accurate arithmetic performed in silicon is the culmination of several layers of technology and artistry. In the preceding chapters, we discussed the primitive operations and schemes on which implementations of arithmetic are based: bit manipulation, floating-point representation, and rounding. The following chapters collect some of the commonly used algorithms and methods that employ these primitives in implementing the elementary operations of addition, multiplication, division, and square root extraction. The application of these methods in the design of a commercial floating-point unit will be illustrated in Part V.

Addition, the most basic of the arithmetic operations, is modeled at the level of bit vectors in Chap. 8, applying the foregoing general theory. From the RTL designer's perspective, the implementation of integer addition is a relatively simple problem. We limit our treatment of this topic, therefore, to a brief introduction, focusing instead on some basic techniques commonly used in the construction of floating-point adders. The opposite is true of multiplication: the design of an integer multiplier presents a more interesting challenge than its application to floating-point multiplication. Thus, in Chap. 9, we limit our attention to an analysis of the former, based on the ubiquitous Booth algorithm [2].

Since division and square root extraction are implemented as sequences of these simpler operations, we model them at the more abstract level of rational numbers, treating addition and multiplication as primitives. Two general approaches form the basis of most implementations of division: (1) digit recurrence, which generates a fixed number of quotient bits on each iteration and subtracts a corresponding multiple of the divisor from the current remainder, and (2) multiplicative methods based on a convergent sequence of approximations of the reciprocal of the divisor derived by Newton's iterative method. Both approaches involve recurrence relations that may be adapted to the computation of the square root, and each has been used in a wide range of variations driven by application requirements and technological constraints. In Chaps. 10 and 11, we present and analyze representative instances of both.

Chapter 8
Addition

The problem of computer addition comprises two distinct tasks: (1) the design of an integer adder, and (2) its application to the addition of floating-point numbers. Integer addition is a nontrivial but relatively simple operation that can be performed for ordinary bit-widths within a single clock cycle. Consequently, this operation is generally treated by the RTL designer as a primitive operation to be implemented by a logic synthesis tool. This amounts to a selection from a library of pre-defined adder modules, based on width and timing requirements. Thus, our treatment of this topic, Sect. 8.1, is limited to a brief introduction, focusing on basic concepts that are relevant to later chapters. The remaining two sections of this chapter deal with optimization techniques that are commonly used in the normalization and rounding of floating-point sums.

8.1 Bit Vector Addition

In this section, we explore the gate-level implementation of bit vector addition. As a first step, we consider the simple 2-gate module of Fig. 8.1, known as a *half adder*. According to the first lemma below, the outputs of this module may be interpreted as the 2-bit sum of its 1-bit inputs.

Lemma 8.1 *If u and v are 1-bit vectors, then*

$$u + v = \{1'(u \mathbin{\&} v), 1'(u \mathbin{\hat{}} v)\}.$$

Proof The equation may be checked exhaustively, i.e., for all 4 possible combinations of u and v. □

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_8

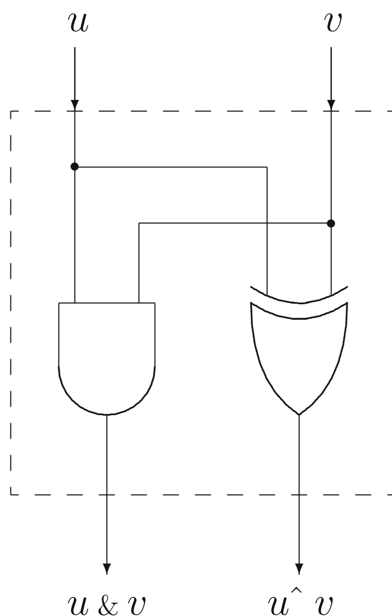


Fig. 8.1 Half adder

The *propagate* and *generate* vectors of two n-bit summands x and y are defined as

$$p = x \ \hat{} \ y$$

and

$$g = x \ \& \ y,$$

respectively. Obviously, these vectors may be computed by a circuit consisting of n half adders, as shown in Fig. 8.2. The following lemma gives a reformulation of $x + y$ in terms of p and g.

Lemma 8.2 *Given $n \in \mathbb{N}$ and n-bit vectors x and y, let $p = x \ \hat{} \ y$ and $g = x \ \& \ y$. Then*

$$x + y = p + 2g.$$

Proof By Lemmas 2.4 and 2.32,

$$\begin{aligned} x + y &= x[n-1:0] + y[n-1:0] \\ &= 2^{n-1}(x[n-1] + y[n-1]) + x[n-2:0] + y[n-2:0]. \end{aligned}$$

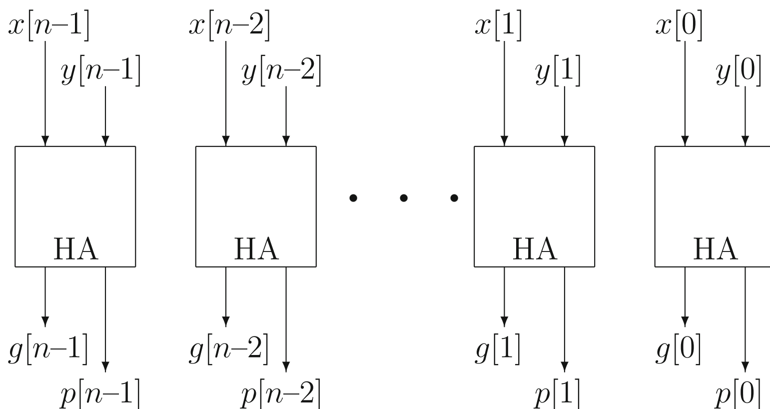


Fig. 8.2 Propagate and generate vectors

Appealing to induction, we may assume that

$$x[n-2:0] + y[n-2:0] = x[n-2:0] \mathbin{\char`^} y[n-2:0] + 2(x[n-2:0] \mathbin{\&} y[n-2:0]),$$

which reduces, by Lemma 3.6, to $p[n-2:0] + 2g[n-2:0]$. By Lemma 8.1,

$$\begin{aligned} x[n-1] + y[n-1] &= \{x[n-1] \mathbin{\&} y[n-1], x[n-1] \mathbin{\char`^} y[n-1]\} \\ &= \{g[n-1], p[n-1]\} \\ &= 2g[n-1] + p[n-1]. \end{aligned}$$

Thus, by Lemma 2.32,

$$\begin{aligned} x + y &= 2^{n-1}(2g[n-1] + p[n-1]) + p[n-2:0] + 2g[n-2:0] \\ &= 2(2^{n-1}g[n-1] + g[n-2:0]) + (2^{n-1}p[n-1] + p[n-2:0]) \\ &= 2g[n-1:0] + p[n-1:0] \\ &= p + 2g. \end{aligned}$$

□

In order to represent $x + y$ as a single vector, we shall require a more complex module, known as a *full adder*, composed of two half adders and an or-gate, as shown in Fig. 8.3. The outputs of the full adder may be readily computed as

$$u \mathbin{\char`^} v \mathbin{\char`^} w$$

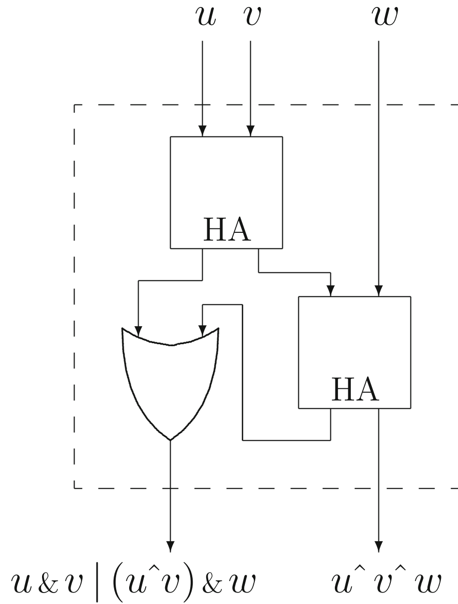


Fig. 8.3 Full adder

and

$$u \mathbin{\&} v \mid (u \mathbin{\hat{}} v) \mathbin{\&} w = u \mathbin{\&} v \mid u \mathbin{\&} w \mid v \mathbin{\&} w.$$

Its arithmetic functionality is characterized as follows.

Lemma 8.3 *If u, v, and w are 1-bit vectors, then*

$$u + v + w = \{1'(u \mathbin{\&} v \mid u \mathbin{\&} w \mid v \mathbin{\&} v), 1'(u \mathbin{\hat{}} v \mathbin{\hat{}} w)\}.$$

Proof The equation may be checked exhaustively, i.e., for all eight possible combinations of u, v, and w. □

Thus, a full adder computes the 2-bit sum of three 1-bit inputs. Replacing the half adders of Fig. 8.2 with full adders results in the *3:2 compressor* of Fig. 8.4, also known as a *carry-save adder*, which reduces a sum of three vectors to a sum of two.

Lemma 8.4 *Given $n \in \mathbb{N}$ and n-bit vectors x, y, and z, let*

$$a = x \mathbin{\hat{}} y \mathbin{\hat{}} z$$

and

$$b = x \mathbin{\&} y \mid x \mathbin{\&} z \mid y \mathbin{\&} z.$$

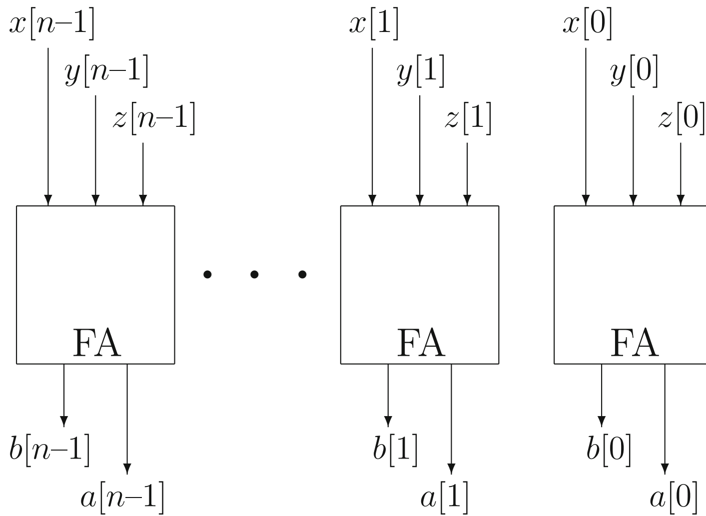


Fig. 8.4 3:2 compressor

Then

$$x + y + z = a + 2b.$$

Proof By Lemmas 2.4 and 2.32,

$$\begin{aligned}
&x + y + z \\
&= x[n–1 : 0] + y[n–1 : 0] + z[n–1 : 0] \\
&= 2^{n-1}(x[n–1] + y[n–1] + z[n–1]) + x[n–2 : 0] + y[n–2 : 0] + z[n–2 : 0].
\end{aligned}$$

By induction, we may assume that

$$x[n–2 : 0] + y[n–2 : 0] + z[n–2 : 0] = a[n–2 : 0] + 2b[n–2 : 0],$$

and by Lemma 8.3,

$$\begin{aligned}
&x[n–1] + y[n–1] + z[n–1] \\
&= 2(x[n–1] \mathbin{\&} y[n–1] \mid x[n–1] \mathbin{\&} z[n–1] \mid y[n–1] \mathbin{\&} z[n–1]) \\
&\quad + x[n–1] \mathbin{\hat{}} y[n–1] \mathbin{\hat{}} z[n–1].
\end{aligned}$$

By Lemma 3.7,

$$x[n–1] \mathbin{\hat{}} y[n–1] \mathbin{\hat{}} z[n–1] = a[n–1]$$

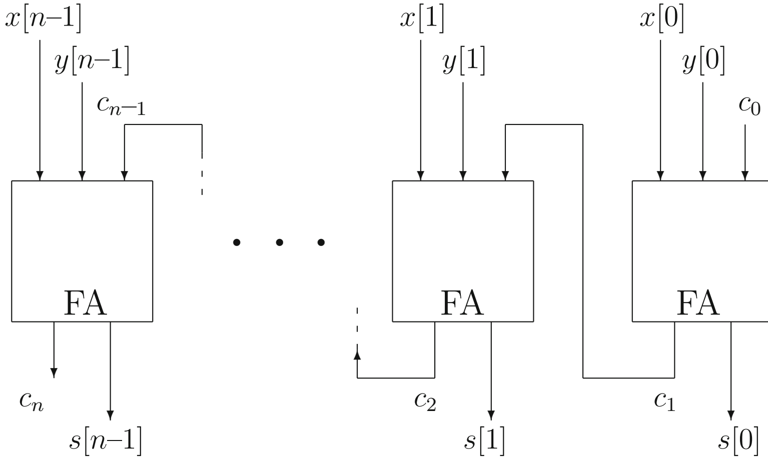


Fig. 8.5 Ripple-carry adder

and

$$x[n–1] \mathbin{\&} y[n–1] \mid x[n–1] \mathbin{\&} z[n–1] \mid y[n–1] \mathbin{\&} z[n–1] = b[n–1].$$

Thus, invoking Lemma 2.32, we have

$$\begin{aligned} x + y + z &= 2^{n-1}(2b[n-1] + a[n-1]) + a[n–2 : 0] + 2b[n–2 : 0] \\ &= (2^{n-1}a[n–1] + a[n–2 : 0]) + 2(2^{n-1}b[n–1] + b[n–2 : 0]) \\ &= a[n–1 : 0] + 2b[n–1 :: 0] \\ &= a + 2b. \end{aligned}$$

□

The module displayed in Fig. 8.5 is constructed from the same hardware as the 3:2 compressor, but the third input $z[k]$ of each adder is eliminated and replaced by c_k, the carry bit generated at index $k - 1$. The resulting circuit, known as a *ripple-carry adder* (RCA), produces the sum $x + y + c_0$ of two n-bit vectors x and y and a carry-in bit c_0, represented as a single n-bit vector s and a carry-out c_n.

Lemma 8.5 *Given $n \in \mathbb{N}$, let x and y be n-bit vectors and let $c_0 = \{0, 1\}$. For $k = 0, \ldots, n - 1$, let*

$$c_{k+1} = x[k] \mathbin{\&} y[k] \mid x[k] \mathbin{\&} c_k \mid y[k] \mathbin{\&} c_k.$$

Let s be the n-bit vector defined by

$$s[k] = x[k] \mathbin{\hat{}} y[k] \mathbin{\hat{}} c_k,$$

for $k = 0, \ldots, n-1$. *Then*

$$x + y + c_0 = \{1'c_n, n's\}.$$

Proof We shall show, by induction on k, that for $0 \leq k \leq n$,

$$x[k\text{–}1:0] + y[k\text{–}1:0] + c_0 = 2^k c_k + s[k\text{–}1:0] = \{1'c_k, s[k\text{–}1:0]\}.$$

The claim is trivial for $k = 0$; assume that it holds for some k, $0 \leq k < n$. Then by Lemmas 2.32 and 8.3,

$$\begin{aligned}
x[k:0] + y[k:0] + c_0 &= 2^k(x[k] + y[k]) + x[k\text{–}1:0] + y[k\text{–}1:0] \\
&= 2^k(x[k] + y[k]) + 2^k c_k + s[k\text{–}1:0] \\
&= 2^k(x[k] + y[k] + c_k) + s[k\text{–}1:0] \\
&= 2^k\{c_{k+1}, s[k]\} + s[k\text{–}1:0] \\
&= 2^{k+1}c_{k+1} + 2^k s[k] + s[k\text{–}1:0] \\
&= 2^{k+1}c_{k+1} + s[k:0].
\end{aligned}$$

□

Since the preceding carry bit c_{k-1} is required for the computation of c_k and $s[k]$, and each full adder incurs two gate delays, the execution time of a RCA of width n is $2n$ gate delays. In order to improve the efficiency of bit vector addition, some degree of parallelism must be introduced in the computation of the carry bits.

Note that the recurrence formula of Lemma 8.5 may be written as

$$c_{k+1} = g_k \mid p_k \,\&\, c_k,$$

where

$$g_k = x[k] \,\&\, y[k]$$

and

$$p_k = x[k] \,\hat{}\, y[k],$$

and successive carry bits may thus be computed as follows:

$$\begin{aligned}
c_{k+2} &= g_{k+1} \mid p_{k+1} \,\&\, c_{k+1} \\
&= g_{k+1} \mid p_{k+1} \,\&\, g_k \mid p_{k+1} \,\&\, p_k \,\&\, c_k, \\
c_{k+3} &= g_{k+2} \mid p_{k+2} \,\&\, c_{k+2}
\end{aligned}$$

$$
\begin{aligned}
&= g_{k+2} \mid p_{k+2} \mathbin{\&} g_{k+1} \mid p_{k+2} \mathbin{\&} p_{k+1} \mathbin{\&} g_k \mid p_{k+2} \mathbin{\&} p_{k+1} \mathbin{\&} p_k \mathbin{\&} c_k, \\
c_{k+4} &= g_{k+3} \mid p_{k+3} \mathbin{\&} c_{k+3} \\
&= g_{k+3} \mid p_{k+3} \mathbin{\&} g_{k+2} \mid p_{k+3} \mathbin{\&} p_{k+2} \mathbin{\&} g_{k+1} \mid p_{k+3} \mathbin{\&} p_{k+2} \mathbin{\&} p_{k+1} \mathbin{\&} g_k \mid \\
&\qquad p_{k+3} \mathbin{\&} p_{k+2} \mathbin{\&} p_{k+1} \mathbin{\&} p_k \mathbin{\&} c_k, \\
&\ldots
\end{aligned}
$$

This observation is the basis of the design of a circuit known as a *carry-look-ahead* adder (CLA), which computes the sum of two n-bit vectors in constant time, independent of n. In fact, this may be achieved in as few as four gate delays:

(1) 1 gate delay to compute the generate and propagate bits, g_k and p_k;
(2) 2 gate delays to compute the carry bits c_k, using the above equations;
(3) 1 gate delay to compute the sum bits, $s[k] = x[k] \mathbin{\hat{}} y[k] \mathbin{\hat{}} c_k$.

However, this design is impractical for all but very small values of n, since the number of gates required increases quadratically with n and the fan-in (number of inputs to each gate) increases linearly. In practice, this technique is used only in very narrow adders, which are connected in series to add wider vectors. The optimal width of a link in such a chain is a function of the propagation delays and other characteristics of the underlying technology, but the most common width is 4.

Example We shall construct an adder of width $n = 4m$, with the same interface as the RCA of Lemma 8.5, as a series of m identical hardware modules. As displayed in Fig. 8.6, for $j = 0, \ldots, m-1$, the inputs of the jth module are the 4-bit operand slices $x[4j+3 : 4j]$ and $y[4j+3 : 4j]$ and the carry out c_{4j} of the preceding module. Its outputs are the corresponding slice of the sum, $s[4j+3 : 4j]$, and the carry-out c_{4j+4}. The computation of this carry-out is based on the above expression for c_{k+4}. Substituting $4j$ for k, we may write this equation as

$$c_{4j+4} = G_j \mid P_j \mathbin{\&} c_{4j},$$

where

$$
\begin{aligned}
G_j = g_{4j+3} \mid p_{4j+3} \mathbin{\&} g_{4j+2} \mid p_{4j+3} \mathbin{\&} p_{4j+2} \mathbin{\&} g_{4j+1} \mid p_{4j+3} \\
\mathbin{\&} p_{4j+2} \mathbin{\&} p_{4j+1} \mathbin{\&} g_{4j}
\end{aligned}
$$

and

$$P_j = p_{4j+3} \mathbin{\&} p_{4j+2} \mathbin{\&} p_{4j+1} \mathbin{\&} p_{4j}.$$

The module consists of four components, which perform the following computations:

(1) For $k = 4j, \ldots, 4j+3$, p_k and g_k are computed by a half adder.

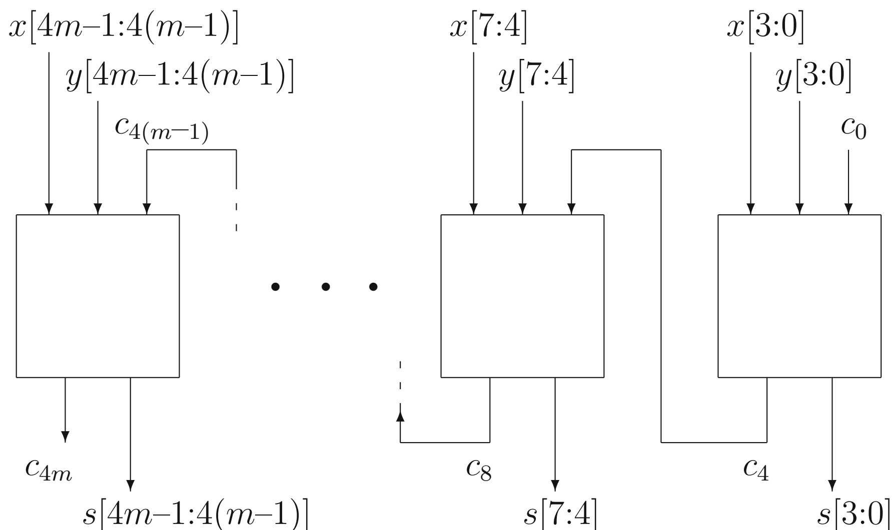


Fig. 8.6 Carry-look-ahead adder

(2) G_j and P_j are derived from the outputs of (1).
(3) c_{4j+4} is derived from G_j, P_j, and c_{4j}.
(4) $s[4j + 3 : 4j]$ is computed from the inputs by a 4-bit RCA or CLA.

Steps (1) and (2) are performed concurrently by each module during an initial period of several gate delays. Steps (3) and (4) may then be performed by each module once the carry-out of the preceding module is available. The critical feature of this design is that successive carry bits are computed two gate delays apart, without waiting for the adder outputs. Consequently, for large n, the total number of gate delays is approximately $2m = n/2$, an improvement over the RCA by a factor of 4.

8.2 Leading Zero Anticipation

We turn now to the problem of adding two numbers that are encoded in a floating-point format. The following procedure represents a naive approach to the design of a floating point adder:

1. Compare the exponent fields of the summands to determine the right shift necessary to align the significands;
2. Perform the required right shift on the significand field that corresponds to the lesser exponent;
3. Add (or subtract) the aligned significands, together with the appropriate rounding constant;

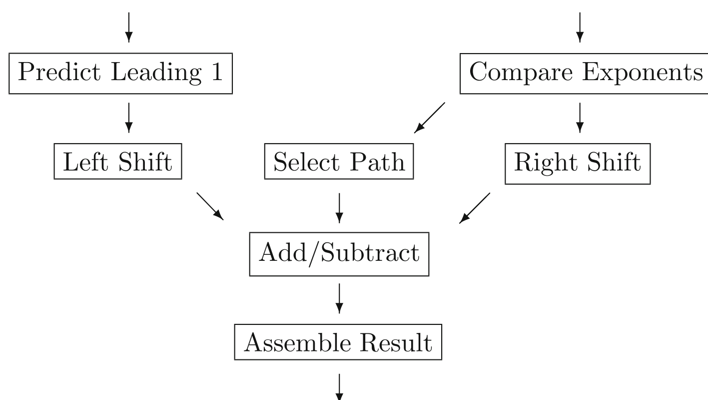


Fig. 8.7 Split-path adder

4. Determine the left shift required to normalize the result;
5. Perform the left shift and adjust the exponent accordingly;
6. Compute the final result by assembling the sign, exponent, and significand fields.

It is possible, however, to reduce the latency of floating-point addition by executing some of these operations in parallel. While a large left shift may be required (in the case of subtraction, if massive cancellation occurs), and a large right shift may be required (if the exponents are vastly different), only one of these possibilities will be realized for any given pair of inputs. Thus, as illustrated in Fig. 8.7, an efficient adder typically includes two data paths, called the *near* and *far* paths. On the near path, the sum is computed under the assumption that an effective subtraction is to be performed and the exponents differ by at most 1. Thus, the summands are aligned quickly, but time is allocated to Steps (4) and (5). On the far path, which handles the remaining case, Steps (1) and (2) are time-consuming, but the sum is easily normalized. A concurrent analysis of the exponents determines which of these paths is actually used to produce the final result.

In order for the operation to be effectively pipelined without duplicating the adder hardware, the paths must merge before the addition is performed. This is made possible by a technique known as *leading zero anticipation*, which allows the left shift (on the near path) to be determined, and perhaps even performed, in advance of the subtraction. Consequently, steps (4) and (5) of the near path may be executed concurrently with steps (1) and (2) of the far path. Meanwhile, the exponent analysis is performed in time to select the inputs to the adder.

Subtraction of bit vectors is naturally implemented as an addition that is guaranteed to overflow. Let a and b be n-bit vectors with $s = a + b > 2^n$. Our objective is to predict the location of the leading one of the sum, i.e, the greatest $i < n$ such that $s[i] = 1$, or $expo(s[n-1:0])$. Although the precise computation of this index is in general as complex as the addition itself, requiring an execution time that is linear in n, a useful approximate solution may be obtained more quickly. We shall compute, in constant time (independent of a, b, and n), a positive integer

Σ	P	P	P	P	P	P	G	K	K	K	P	K	G	P	P
a	1	0	1	1	0	1	1	0	0	0	1	0	1	0	0
b	0	1	0	0	1	0	1	0	0	0	0	0	1	1	1
s	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1
p	1	1	1	1	1	1	0	0	0	0	1	0	0	1	1
g	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
k	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0

Fig. 8.8 Leading zero anticipation

w such that $expo(s[n-1:0])$ is either $expo(w)$ or $expo(w)-1$. As we shall see in Chap. 17, this technique may be combined with a leading zero counter that runs in logarithmic time.

We begin with an informal motivating discussion; a formal solution is given by Lemma 8.6 below. The technique is based on the propagate and generate vectors discussed in Sect. 8.1, $p = a \mathbin{\hat{}} b$ and $g = a \mathbin{\&} b$. We also define the *kill* vector, $k = {\sim}a \mathbin{\&} {\sim}b$. As illustrated in Fig. 8.8, for each index i, exactly one of $p[i]$, $g[i]$, and $k[i]$ is asserted, and we associate i with one of the symbols P, G, and K accordingly, creating a string of symbols $\Sigma = \sigma_{n-1} \ldots \sigma_1 \sigma_0$, where

$$\sigma_i = \begin{cases} P \text{ if } p_i = 1 \\ G \text{ if } g_i = 1 \\ K \text{ if } k_i = 1. \end{cases}$$

We shall identify an index j such that the leading one of $s[n-1:0]$ occurs at either j or $j-1$. While the computation is actually performed in constant time, it is instructive to view the search as a left-to-right traversal of Σ. Since the hypothesis $s > 2^n$ implies $a[n-1] + b[n-1] \geq 1$, we must have $k[n-1] = 0$ and σ_{n-1} is either P or G. If $\sigma_{n-1} = P$, then there must be a carry into index $n-1$, resulting in $s[n-1] = 0$, and we may ignore this index and move to index $n-2$. We continue in this manner until we reach the first occurrence of the symbol G, which we find at some index i. Thus, $\sigma_{n-1} = \sigma_{n-2} = \ldots = \sigma_{i+1} = P$ and $\sigma_i = G$. If σ_{i-1} is either P or G, then the leading one of s must occur at either i or $i-1$ depending on whether a carry is produced at $i-1$, and our search is concluded with index $j = i$. But if $\sigma_{i-1} = K$, then the traversal continues until we reach the index j at which the final K in the string occurs, i.e., $\sigma_{i-1} = \sigma_j = K$ and $\sigma_{j-1} \neq K$. In this case, the leading one must occur at either j or $j-1$, depending on whether a carry is produced at $j-1$.

Thus, we select j as the terminal index of the maximal prefix of the string of the form P^*GK^* (0 or more Ps, followed by a single G, followed by 0 or more Ks). Clearly, this is the maximal $i < n$ for which $\sigma_i \sigma_{i-1}$ is not one of the following combinations: PP, PG, GK, or KK. But this may also be characterized as the maximal $i < n$ such that $w[i] = 1$, where

$$w[i] = {\sim}\big((p[i] \mathbin{\&} p[i{-}1]) \mid (p[i] \mathbin{\&} g[i{-}1]) \mid (g[i] \mathbin{\&} k[i{-}1]) \mid (k[i] \mathbin{\&} k[i{-}1])\big)$$

$$= \sim\big((p[i] \ \& \ (p[i{-}1] \mid g[i{-}1])) \mid ((g[i] \mid k[i]) \ \& \ k[i{-}1])\big)$$
$$= \sim\big((p[i] \ \& \ k[i{-}1]) \mid (p[i] \ \& \ k[i{-}1])\big)$$
$$= \sim(p[i] \wedge k[i{-}1]),$$

that is, the leading one of the vector $w = \sim(p \wedge 2k)[n{-}1 : 0]$.

A more rigorous version of this derivation follows.

Lemma 8.6 *Let a and b be n-bit vectors, where* $n \in \mathbb{Z}^+$, *with* $s = a + b > 2^n$. *Let* $p = a \wedge b$, $k = \sim a[n{-}1 : 0] \ \& \ \sim b[n{-}1 : 0]$, *and*

$$w = \sim(p \wedge 2k)[n{-}1 : 0].$$

Then

$$expo(w) - 1 \leq expo(s[n{-}1 : 0]) \leq expo(w).$$

Proof We shall prove by induction on j that for $0 \leq j < n$, if

(a) $w[n{-}1 : j + 1] = 0$,
(b) $s[n{-}1 : j + 1] = 0$, and
(c) either $a[j : 0] + b[j : 0] \geq 2^{j+1}$ or $k[j] = 1$,

then the above inequalities hold. Since these conditions are clearly satisfied by $j = n - 1$, this is sufficient.

The case $j = 0$ is vacuously true, since (b) implies $s[0] = s[n{-}1 : 0] \neq 0$, and therefore $a[0] + b[0] = s[0] = 1$, contradicting (c).

Assume that (a), (b), and (c) hold for some $j, 0 < j < n$. First suppose $w[j] = 1$. Then $expo(w) = j$ and $p[j] = k[j{-}1]$. If $p[j] = k[j{-}1] = 1$, then

$$\begin{aligned}
&a[j : 0] + b[j : 0] \\
&= 2^j(a[j] + b[j]) + 2^{j-1}(a[j{-}1] + b[j{-}1]) + a[j - 2 : 0] + b[j - 2 : 0] \\
&= 2^j + a[j - 2 : 0] + b[j - 2 : 0] \\
&< 2^j + 2^{j-1} + 2^{j-1} \\
&= 2^{j+1},
\end{aligned}$$

contradicting (c). Therefore, we must have $p[j] = k[j{-}1] = 0$. Since $p[j] = 0$, either $a[j] = b[j] = 0$, in which case

$$s[n{-}1 : 0] = s[j : 0] = a[j{-}1 : 0] + b[j{-}1 : 0],$$

or $a[j] = b[j] = 1$, which implies

$$\begin{aligned} s[n\text{–}1:0] = s[j:0] &= (a[j:0] + b[j:0]) \bmod 2^{j+1} \\ &= (2^j + a[j\text{–}1:0] + 2^j + b[j\text{–}1:0]) \bmod 2^{j+1} \\ &= a[j\text{–}1:0] + b[j\text{–}1:0]. \end{aligned}$$

Since $k[j\text{–}1] = 0$, $2^{j-1} \leq a[j\text{–}1:0] + b[j\text{–}1:0] < 2^{j+1}$, i.e, $2^{j-1} \leq s[n\text{–}1:0] < 2^{j+1}$ and

$$expo(w) - 1 = j - 1 \leq expo(s[n\text{–}1:0]) \leq j = expo(w).$$

Now suppose $w[j] = 0$. Then $p[j] \neq k[j\text{–}1]$. We shall complete the induction by showing that (a), (b), and (c) hold for $j - 1$.

(a) $w[n\text{–}1:j] = 2w[n\text{–}1:j+1] + w[j] = 0 + 0 = 0$.

(c) This holds trivially if $k[j\text{–}1] = 1$, so we may assume $k[j\text{–}1] = 0$ and $p[j] = 1$ Since $k[j] = 0$, we must have $a[j:0] + b[j:0] \geq 2^{j+1}$. It follows that $a[j\text{–}1:0] + b[j\text{–}1:0] \geq 2^j$, for otherwise

$$\begin{aligned} a[j:0] + b[j:0] &= 2^j(a[j] + b[j]) + a[j\text{–}1:0] + b[j\text{–}1:0] \\ &= 2^j + a[j\text{–}1:0] + b[j\text{–}1:0] \\ &< 2^{j+1}. \end{aligned}$$

(b) We shall show that $s[j:0] < 2^j$. If $p[j] = 1$ and $k[j\text{–}1] = 0$, then, as we have shown, $a[j\text{–}1:0] + b[j\text{–}1:0] \geq 2^j$, and therefore

$$\begin{aligned} s[j:0] &= (a[j:0] + b[j:0]) \bmod 2^{j+1} \\ &= (2^j + a[j\text{–}1:0] + b[j\text{–}1:0]) \bmod 2^{j+1} \\ &= (2^{j+1} + (a[j\text{–}1:0] + b[j\text{–}1:0] - 2^j)) \bmod 2^{j+1} \\ &= (a[j\text{–}1:0] + b[j\text{–}1:0] - 2^j) < 2^j. \end{aligned}$$

But if $p[j] = 0$ and $k[j\text{–}1] = 1$, then either $k[j] = 1$ and

$$s[j:0] = a[j:0] + b[j:0] = a[j-2:0] + b[j-2:0] < 2^{j-1} + 2^{j-1} = 2^j$$

or $a[j] = b[j] = 1$ and

$$\begin{aligned} s[j:0] &= (a[j:0] + b[j:0]) \bmod 2^{j+1} \\ &= (2^j + a[j-2:0] + 2^j + b[j-2:0]) \bmod 2^{j+1} \\ &= a[j-2:0] + b[j-2:0] \\ &< 2^j. \end{aligned}$$

In any case, $s[j:0] < 2^j$ implies $s[j] = 0$ and

$$s[n\text{–}1:j] = 2s[n\text{–}1:j+1] + s[j] = 0.$$

□

8.3 Trailing Zero Anticipation

As we saw in Chap. 6, the process of rounding a bit vector often involves determining its degree of exactness. For this purpose, therefore, it is also useful to predict the *trailing zeroes* of a sum, i.e., the least index at which a one occurs. The following lemmas provide methods for computing, in constant time, an integer that has precisely the same trailing zeroes as the sum or difference of two given operands.

The difference $x - y$ of n-bit vectors x and y is naturally computed as a sum, using the identity $\sim y[n\text{–}1:0] = 2^n - y[n\text{–}1:0] - 1$, which leads to the formula

$$(x - y)[n\text{–}1:0] = (x + \sim y[n\text{–}1:0] + 1)[n\text{–}1:0].$$

Thus, we are also interested in computing the trailing one of an incremented sum. This problem admits a particularly simple solution.

Lemma 8.7 *Let a and b be n-bit vectors, where $n \in \mathbb{N}$. For all $k \in \mathbb{N}$, if $k < n$, then*

$$(a + b + 1)[k:0] = 0 \Leftrightarrow \sim(a \mathbin{\hat{}} b)[k:0] = 0.$$

Proof First we consider the case $k = 0$. By Corollary 2.7,

$$(a + b + 1)[0] = (a[0] + b[0] + 1)[0],$$

and exhaustive testing yields

$$(a[0] + b[0] + 1)[0] = \sim(a[0] \mathbin{\hat{}} b[0]).$$

Thus, by Lemmas 3.20 and 3.7,

$$\sim(a[0] \mathbin{\hat{}} b[0]) = \sim(a \mathbin{\hat{}} b)[0] = (a + b + 1)[0].$$

We proceed by induction, assuming $k > 0$. Applying Lemma 2.33, we may assume that

$$(a + b + 1)[0] = \sim(a[0] \mathbin{\hat{}} b[0]) = 0$$

and need only show that

$$(a + b + 1)[k : 1] = 0 \Leftrightarrow \sim(a[n\text{–}1 : 0] \wedge b[n\text{–}1 : 0])[k : 1] = 0.$$

Let $a' = \lfloor a/2 \rfloor$ and $b' = \lfloor b/2 \rfloor$. By inductive hypothesis,

$$(a' + b' + 1)[k\text{–}1 : 0] = 0 \Leftrightarrow \sim(a' \wedge b')[k\text{–}1 : 0] = 0.$$

By Lemmas 2.12, 3.19, and 3.6,

$$\begin{aligned}\sim(a \wedge b)[k : 1] &= \sim(a[k : 1] \wedge b[k : 1]) \\ &= \sim(a'[k\text{–}1 : 0] \wedge b'[k\text{–}1 : 0]) \\ &= \sim(a' \wedge b')[k\text{–}1 : 0].\end{aligned}$$

Therefore, it suffices to show that

$$(a + b + 1)[k : 1] = (a' + b' + 1)[k\text{–}1 : 0].$$

Since

$$(a + b + 1)[0] = (a[0] + b[0] + 1)[0] = 0,$$

$a[0] + b[0] + 1$ is even, which implies $a[0] + b[0] + 1 = 2$. Thus,

$$a + b + 1 = (2a' + a[0]) + (2b' + b[0]) + 1 = 2(a' + b' + 1)$$

and by Lemma 2.14,

$$(a + b + 1)[k : 1] = (a' + b' + 1)[k - 1 : 0].$$

□

Our next lemma is applicable to both addition and subtraction, but involves a somewhat more complicated computation.

Lemma 8.8 *Let a and b be n-bit vectors, where $n \in \mathbb{N}$, and let $c \in \{0, 1\}$. Let*

$$\tau = a \wedge b \wedge (2(a \mid b) \mid c).$$

Then for all $k \in \mathbb{N}$, if $k < n$, then

$$(a + b + c)[k : 0] = 0 \Leftrightarrow \tau[k : 0] = 0.$$

Proof Exhaustive testing yields

$$(a[0] + b[0] + c)[0] = a[0] \wedge b[0] \wedge c,$$

and hence, by Corollary 2.7 and Lemmas 3.7, 2.29, and 2.23,

$$(a + b + 1)[0] = (a[0] + b[0] + 1)[0] = a[0] \mathbin{\char`^} b[0] \mathbin{\char`^} c = \tau[0].$$

This establishes the case $k = 0$. We proceed by induction as in the proof of Lemma 8.7. Again, by Lemma 2.33, we may assume that

$$(a + b + c)[0] = \tau[0] = 0$$

and need only show that

$$(a + b + c)[k:1] = 0 \Leftrightarrow \tau[k:1] = 0.$$

Let $a' = \lfloor a/2 \rfloor$, $b' = \lfloor b/2 \rfloor$, $c' = a[0] \mid b[0]$, and

$$\tau' = a' \mathbin{\char`^} b' \mathbin{\char`^} (2(a' \mid b') \mid c').$$

By inductive hypothesis,

$$(a' + b' + c')[k{-}1:0] = 0 \Leftrightarrow \tau'[k{-}1:0] = 0.$$

By Lemmas 3.19 and 2.12,

$$\begin{aligned} \tau[k:1] &= a[k:1] \mathbin{\char`^} b[k:1] \mathbin{\char`^} ((2(a \mid b))[k:1] \\ &= a'[k{-}1:0] \mathbin{\char`^} b'[k{-}1:0] \mathbin{\char`^} (a \mid b)[k{-}1:0]. \end{aligned}$$

The same lemmas in combination with Lemma 2.33, Corollary 3.11, and Lemma 2.15, yield

$$\begin{aligned} (a \mid b)[k{-}1:0] &= 2(a \mid b)[k{-}1:1] + (a \mid b)[0] \\ &= 2(a \mid b)[k{-}1:1] + c' \\ &= 2(a \mid b)[k{-}1:1] \mid c' \\ &= 2(a' \mid b')[k{-}2:0] \mid c' \\ &= (2(a' \mid b'))[k{-}1:0] \mid c' \\ &= (2(a' \mid b') \mid c')[k{-}1:0]. \end{aligned}$$

Thus,

$$\tau[k:1] = a'[k{-}1:0] \mathbin{\char`^} b'[k{-}1:0] \mathbin{\char`^} (2(a' \mid b') \mid c')[k{-}1:0] = \tau'[k{-}1:0],$$

and it suffices to show that

$$(a + b + c)[k : 1] = (a' + b' + c')[k–1 : 0].$$

Since

$$(a + b + c)[0] = (a[0] + b[0] + c)[0] = 0,$$

$a[0] + b[0] + c$ is even, which implies $a[0] + b[0] + c = 2(a[0] \mid b[0]) = 2c'$. Thus,

$$a + b + c = (2a' + a[0]) + (2b' + b[0]) + c = 2(a' + b' + c')$$

and by Lemma 2.14,

$$(a + b + c)[k : 1] = (a' + b' + c')[k - 1 : 0].$$

□

Chapter 9
Multiplication

While the RTL implementation of integer multiplication is more complex than that of integer addition, the extended problem of floating-point multiplication does not present any significant difficulties that have not already been addressed, and is, in fact, much simpler than floating-point addition. The focus of this chapter, therefore, is the multiplication of natural numbers. All of our results can be readily extended to signed integers.

Let $x \in \mathbb{N}$ and $y \in \mathbb{N}$. We shall refer to x and y as the *multiplicand* and the *multiplier*, respectively. A natural approach to the computation of the product xy begins with the bit-wise decomposition of the multiplier provided by Corollary 2.38:

$$y = (\beta_{w-1} \cdots \beta_0)_2 = \sum_{i=0}^{w-1} 2^i \beta_i, \tag{9.1}$$

where $0 \leq y < 2^w$ and for $i = 0, \ldots, w-1$, $\beta_i = y[i]$. The product may then be computed as

$$xy = \sum_{i=0}^{w-1} 2^i \beta_i x.$$

Thus, the computation is reduced to the summation of at most w nonzero terms, called *partial products*, each of which is derived by an appropriate shift of x. In practice, this summation is performed with the use of a tree of compressors similar to the 3:2 compressor shown in Fig. 8.4, which reduces the number of addends to 2 so that only a single carry-propagate addition is involved in the computation of the product. It is clear that two 3:2 compressors may be combined to form a 4:2 compressor, and that 2^{k-2} 4:2 compressors may be used to reduce a sum of 2^k terms to 2^{k-1} in constant time. Consequently, the hardware needed to compress w terms to two grows linearly with w, and the required time grows logarithmically.

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_9

i	11	10	9	8	7	6	5	4	3	2	1	0
$y[i]$	0	1	1	1	0	0	1	1	1	1	1	0
β_i	1	0	0	−1	0	1	0	0	0	0	−1	0

Fig. 9.1 Radix-2 Booth encoding of $y = (011100111110)_2$

Naturally, any reduction in the number of partial products generated in a multiplication would tend to reduce the latency of the operation. Most modern multipliers achieve this objective through some version of a technique discovered by A. D. Booth in the early days of computer arithmetic [2]. After a brief discussion of Booth's original algorithm (Sect. 9.1), we shall present a popular refinement known as the *radix-4 modified Booth algorithm* [15] (Sect. 9.2), which limits the number of partial products to half the multiplier width. Each of the three subsequent sections contains a variant of this algorithm.

9.1 Radix-2 Booth Encoding

Booth encoding is based on the observation that if we allow -1, along with 0 and 1, as a value of the digit β_i in (9.1), then the representation is no longer unique. Thus, we may seek to minimize the number of nonzero digits and consequently the number of partial products in the expression for xy, at the expense of introducing a negation along with the shift of x in the case $\beta_i = -1$.

In fact, as illustrated for a 12-bit vector y in Fig. 9.1, any maximal uninterrupted sequence of 1s in the binary expansion of y,

$$y[k] = y[k+1] = \ldots = y[\ell-1] = 1,$$

may be replaced with as few as two nonzero entries,

$$\beta_k = -1, \beta_{k+1} = \ldots = \beta_{\ell-1} = 0, \beta_\ell = 1,$$

using the identity

$$\sum_{i=k}^{\ell-1} 2^i = 2^\ell - 2^k.$$

Thus, the decomposition $\beta_i = y[i]$ is replaced by the formula

$$\beta_i = \begin{cases} 1 \text{ if } y[i] = 0 \text{ and } y[i-1] = 1 \\ -1 \text{ if } y[i] = 1 \text{ and } y[i-1] = 0 \\ 0 \text{ if } y[i] = y[i-1], \end{cases}$$

or more simply,

$$\beta_i = y[i-1] - y[i].$$

The correctness of this encoding is easily established: if y is an n-bit vector, then since $y[n] = y[-1] = 0$,

$$\begin{aligned}
\sum_{i=0}^{n} 2^i \beta_i &= \sum_{i=0}^{n} 2^i y[i-1] - \sum_{i=0}^{n} 2^i y[i] \\
&= \sum_{i=-1}^{n-1} 2^{i+1} y[i] - \sum_{i=0}^{n} 2^i y[i] \\
&= \sum_{i=0}^{n} 2^{i+1} y[i] - \sum_{i=0}^{n} 2^i y[i] \\
&= \sum_{i=0}^{n} 2^i y[i] \\
&= y.
\end{aligned}$$

Clearly, the reduction in partial products afforded by this scheme is greatest for arguments containing long strings of 1s, while in the worst case (alternating 0s and 1s), all digits are nonzero, and of course, the number of such digits cannot be predicted. Consequently, any benefit is limited to designs that generate and accumulate partial products sequentially, as was common in early computing, when execution speed was readily sacrificed to conserve die area. Modern multiplier architectures, however, employ compression trees that combine the partial products in a single cycle. An effective multiplier encoding scheme today must reduce the depth of such a tree in a consistent and predictable way.

9.2 Radix-4 Booth Encoding

As a notational convenience, we shall assume that x and y are bit vectors of widths $n-1$ and $2m-1$, respectively. Our objective is an efficient computation of xy as a sum of m partial products. Conceptually, the multiplier y is partitioned into m 2-bit slices, $y[2i+1:2i]$, $i = 0, \ldots, m-1$. We seek to define an integer digit θ_i corresponding to each slice such that

$$y = \sum_{i=0}^{m-1} 2^{2i} \theta_i.$$

$y[2i{+}1{:}2i{-}1]$	$\beta_{2i+1}\beta_{2i}$	θ_i
000	0 0	0
001	0 1	1
010	1 −1	1
011	1 0	2
100	−1 0	−2
101	−1 1	−1
110	0 −1	−1
111	0 0	0

Fig. 9.2 Radix-4 Booth encoding

The standard radix-4 representation,

$$\theta_i = 2y[2i{+}1] + y[2i] = y[2i + 1 : 2i],$$

offers no advantage over the ordinary binary representation, since the case $\theta_i = 3$ effectively involves two partial products rather than one. Instead, we define θ_i as a combination of two successive radix-2 Booth digits:

$$\begin{aligned}\theta_i &= 2\beta_{2i+1} + \beta_{2i}\\ &= 2(y[2i] - y[2i{+}1]) + (y[2i{-}1] - y[2i])\\ &= y[2i{-}1] + y[2i] - 2y[2i{+}1].\end{aligned}$$

Since β_{2i+1} and β_{2i} cannot be nonzero and equal, these digits are confined to the range $-2 \leq \theta_i \leq 2$, and we have an expression for y as a sum of at most m nonzero terms, each with absolute value a power of 2, as summarized in Fig. 9.2.

Definition 9.1 For $y \in \mathbb{N}$ and $i \in \mathbb{N}$,

$$\theta_i(y) = y[2i{-}1] + y[2i] - 2y[2i{+}1].$$

Lemma 9.1 *If $m \in \mathbb{Z}^+$ and y is a bit vector of width $2m - 1$, then*

$$y = \sum_{i=0}^{m-1} 2^{2i}\theta_i(y).$$

Proof We shall prove, by induction, that for $0 \leq k \leq m$,

$$y[2k{-}1 : 0] = \sum_{i=0}^{k-1} 2^{2i}\theta_i + 2^{2k}y[2k{-}1],$$

where $\theta_i = \theta_i(y)$. The claim is trivial for $k = 0$. Assuming that it holds for some $k < m$, we have

$$\begin{aligned}
y[2k+1:0] &= 2^{2k}y[2k+1:2k] + y[2k{-}1:0] \\
&= 2^{2k}(2y[2k{+}1] + y[2k]) + \sum_{i=0}^{k-1} 2^{2i}\theta_i + 2^{2k}y[2k{-}1] \\
&= 2^{2k}(2y[2k{+}1] + y[2k] + y[2k{-}1]) + \sum_{i=0}^{k-1} 2^{2i}\theta_i \\
&= 2^{2k}(4y[2k{+}1] + \theta_k) + \sum_{i=0}^{k-1} 2^{2i}\theta_i \\
&= \sum_{i=0}^{k} 2^{2i}\theta_i + 2^{2(k+1)}y[2(k+1){-}1],
\end{aligned}$$

which completes the induction. In particular, substituting m for k, we have

$$y = y[2m{-}1:0] = \sum_{i=0}^{m-1} 2^{2i}\theta_i + 2^{2m}y[2m{-}1] = \sum_{i=0}^{m-1} 2^{2i}\theta_i.$$

□

Our goal is to compute

$$xy = x\sum_{i=0}^{m-1} 2^{2i}\theta_i = \sum_{i=0}^{m-1} 2^{2i}x\theta_i.$$

Each term of this sum will correspond to a row in an array of m partial products of width $2m + n$, which are constructed by means of a 5-to-1 multiplexer. If $\theta_i \in \{0, 1, 2\}$, then the ith term may be represented by the n-bit vector $x\theta$ shifted $2i$ bits to the left. If $\theta_i \in \{-1, -2\}$, then in place of $x\theta$ we use the complement

$$\sim(-x\theta)[n-1:0] = 2^n + x\theta - 1. \tag{9.2}$$

The challenge in constructing the partial product array is to account for the discrepancy between this and the desired value $x\theta$.

Lemma 9.2 *Let x and y be bit vectors of widths $n-1$ and $2m-1$, respectively, where $m \in \mathbb{Z}^+$ and $n \in \mathbb{Z}^+$. For $i = 0, \dots, m-1$, let $\theta_i = \theta_i(y)$,*

$$\sigma_i = \begin{cases} 0 \text{ if } \theta_i \geq 0 \\ 1 \text{ if } \theta_i < 0, \end{cases}$$

$$\bar{\sigma}_i = 1 - \sigma_i,$$

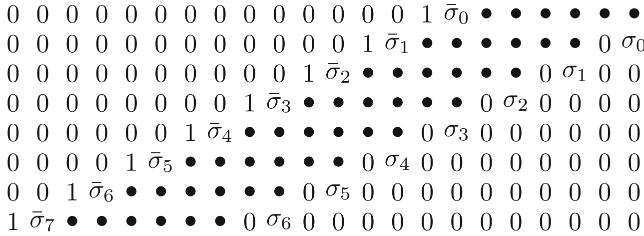


Fig. 9.3 Radix-4 partial product array

$$B_i = \begin{cases} x\theta & \textit{if}\ \sigma_i = 0 \\ \sim(-x\theta)[n-1:0] & \textit{if}\ \sigma_i = 1, \end{cases}$$

and

$$pp_i = \begin{cases} \{2(m-1)\,'\,0, 1, \bar{\sigma}_0, n\,'\,B_0\} & \textit{if}\ i = 0 \\ \{2(m-i-1)\,'\,0, 1, \bar{\sigma}_i, n\,'\,B_i, 0, \sigma_{i-1}, 2(i-1)\,'\,0\} & \textit{if}\ i > 0. \end{cases}$$

Then

$$2^n + \sum_{i=0}^{m-1} pp_i = 2^{n+2m} + xy.$$

Proof As a visual aid, the array of partial products pp_i is depicted for the case $m = 8$, $n = 6$ in Fig. 9.3. In order to understand the construction of this array, note that according to (9.2), whenever $\sigma_i = 1$, we must correct for the discrepancy between $2^{2i}B_i$ and $2^{2i}x\theta_i$ by adding 2^{2i} and subtracting 2^{2i+n}. The addition is achieved simply by the insertion of σ_i at index $2i$ of pp_{i+1}. (This is not an issue for $i = m-1$ because $y[2m-1] = 0$, which implies $\theta_{m-1} \geq 0$.) The subtraction may be viewed as a two-step process. First, we insert 1s at indices $2i+n$ and $2i+n+1$ of pp_i. The cumulative effect of this is merely a carry-out to index $2m+n$. But the motivation for this step is that the 1 at index $2i+n$ may now be subtracted off in the case $\sigma_i = 1$. Thus, in the second step, we replace that 1 with $\bar{\sigma}_i = 1 - \sigma_i$.

This strategy underlies the present proof. According to the definitions of pp_i and $\bar{\sigma}_i$, the left-hand side of the conclusion of the lemma is

$$2^n + \sum_{i=0}^{m-1}\left(2^{n+1+2i} + 2^{n+2i}(1-\sigma_i) + 2^{2i}B_i\right) + \sum_{i=1}^{m-1} 2^{2(i-1)}\sigma_{i-1}.$$

We first consider the constant terms of this sum:

$$\begin{array}{cccccccccccccccccccccc}
0&0&0&0&0&0&0&0&0&0&0&0&0&\bar{\sigma}_0&\sigma_0&\sigma_0&\bullet&\bullet&\bullet&\bullet&\bullet&\bullet\\
0&0&0&0&0&0&0&0&0&0&0&0&1&\bar{\sigma}_1&\bullet&\bullet&\bullet&\bullet&\bullet&\bullet&0&\sigma_0\\
0&0&0&0&0&0&0&0&0&0&1&\bar{\sigma}_2&\bullet&\bullet&\bullet&\bullet&\bullet&\bullet&0&\sigma_1&0&0\\
0&0&0&0&0&0&0&0&1&\bar{\sigma}_3&\bullet&\bullet&\bullet&\bullet&\bullet&\bullet&0&\sigma_2&0&0&0&0\\
0&0&0&0&0&0&1&\bar{\sigma}_4&\bullet&\bullet&\bullet&\bullet&\bullet&\bullet&0&\sigma_3&0&0&0&0&0&0\\
0&0&0&0&1&\bar{\sigma}_5&\bullet&\bullet&\bullet&\bullet&\bullet&\bullet&0&\sigma_4&0&0&0&0&0&0&0&0\\
0&0&1&\bar{\sigma}_6&\bullet&\bullet&\bullet&\bullet&\bullet&\bullet&0&\sigma_5&0&0&0&0&0&0&0&0&0&0\\
1&\bar{\sigma}_7&\bullet&\bullet&\bullet&\bullet&\bullet&\bullet&0&\sigma_6&0&0&0&0&0&0&0&0&0&0&0&0
\end{array}$$

Fig. 9.4 Radix-4 partial product array (second version)

$$\begin{aligned}
2^n + \sum_{i=0}^{m-1}(2^{n+1+2i} + 2^{n+2i}) &= 2^n\left(1 + \sum_{i=0}^{m-1}(2^{2i+1} + 2^{2i})\right)\\
&= 2^n\left(1 + \sum_{j=0}^{2m-1} 2^j\right)\\
&= 2^{n+2m}.
\end{aligned}$$

Next, we observe that since $\sigma_{m-1} = 0$, the final term of the sum may be rewritten as

$$\sum_{i=1}^{m-1} 2^{2(i-1)}\sigma_{i-1} = \sum_{i=0}^{m-2} 2^{2i}\sigma_i = \sum_{i=0}^{m-1} 2^{2i}\sigma_i.$$

Thus, Lemma 9.1 yields

$$\begin{aligned}
2^n + \sum_{i=0}^{m-1} pp_i &= 2^{n+2m} + \sum_{i=0}^{m-1}(-2^{n+2i}\sigma_i + 2^{2i}B_i) + \sum_{i=0}^{m-1} 2^{2i}\sigma_i\\
&= 2^{n+2m} + \sum_{i=0}^{m-1} 2^{2i}\left(B_i - (2^n - 1)\sigma_i\right)\\
&= 2^{n+2m} + \sum_{i=0}^{m-1} 2^{2i}x\theta_i\\
&= 2^{n+2m} + xy.
\end{aligned}$$

□

Note that computing a product as an application of Lemma 9.2 requires injecting an extra bit into the partial product array at index n. This requirement can be eliminated through a minor modification of the low-order entry pp_0 as shown in Fig. 9.4.

Lemma 9.3 *Let x and y be bit vectors of widths $n-1$ and $2m-1$, respectively, where $m \in \mathbb{Z}^+$ and $n \in \mathbb{Z}^+$. For $i = 0, \ldots, m-1$, let θ_i, σ_i, $\bar{\sigma}_i$, B_i, and pp_i be as defined as in Lemma 9.2 except that*

$$pp_0 = \{(2m\text{–}3)\,'\,0, \bar{\sigma}_0, \sigma_0, \sigma_0, n\,'\,B_0\}.$$

Then

$$\sum_{i=0}^{m-1} pp_i \equiv xy \pmod{2^{n+2m}}.$$

Proof The difference between the two definitions of pp_0 is

$$\begin{aligned}
&(2^{n+2}\bar{\sigma}_0 + 2^{n+1}\sigma_0 + 2^n\sigma_0) - (2^{n+1} + 2^n\bar{\sigma}_0) \\
&= 2^n\,(4(1-\sigma_0) + 2\sigma_0 + \sigma_0 - 2 - (1-\sigma_0)) \\
&= 2^n.
\end{aligned}$$

□

Another common minor optimization is in the determination of the sign bit σ_i. Note that for $i > 0$, both σ_i and σ_{i-1} are required in the construction of pp_i. If we define $\sigma_i = y[2i+1]$, then both of these required bits may be easily extracted from the current slice $y[2i{+}1 : 2i{-}1]$. This may produce a different result in the case $\theta_i = 0$, but the overall sum is unchanged. This is the variation of the algorithm that is used in the multiplier of Chap. 16:

Lemma 9.4 *Let x and y be bit vectors of widths $n-1$ and $2m-1$, respectively, where $m \in \mathbb{Z}^+$ and $n \in \mathbb{Z}^+$. For $i = 0, \ldots, m-1$, let θ_i, σ_i, $\bar{\sigma}_i$, B_i, and pp_i be as defined as in Lemma 9.3 except that*

$$\sigma_i = y[2i+1].$$

Then

$$\sum_{i=0}^{m-1} pp_i \equiv xy \pmod{2^{n+2m}}.$$

Proof It is easily checked that of the eight possible values of the slice $y[2i{+}1 : 2i{-}1]$, the modification produces the same σ_i except when $y[2i{+}1 : 2i{-}1] = (111)_2$, in which case we have $\sigma_i = 1$ and $B_i = 2^n - 1$, whereas the corresponding values computed by the definitions of Lemma 9.2 are $\sigma_i' = B_i' = 0$. When this occurs, we

must have $0 \leq i < 2m - 1$ and the affected partial products are pp_i and pp_{i+1}. If $i > 0$, then the resulting change in pp_i

$$2^{2i}\left(2^n\bar{\sigma}_i' + B_i'\right) - 2^{2i}\left(2^n\bar{\sigma}_i' + B_i'\right) = 2^{2i}\left(0 + (2^n - 1)\right) - 2^{2i}\left(2^n + 0\right)$$
$$= -2^{2i},$$

and if $i = 0$, the result is the same:

$$\begin{aligned}
&\left(2^{n+2}\bar{\sigma}_0' + 2^{n+1}\sigma_0' + 2^n\sigma_0' + B_0'\right) - \left(2^{n+2}\bar{\sigma}_0 + 2^{n+1}\sigma_0 + 2^n\sigma_0 + B_0\right)\\
&= \left(0 + 2^{n+1} + 2^n + (2^n - 1)\right) - \left(2^{n+2} + 0 + 0 + 0\right)\\
&= -1\\
&= -2^{2i}.
\end{aligned}$$

On the other hand, the change in pp_{i+1} is

$$2^{2i}(\sigma_i' - \sigma_i) = 2^{2i}(1 - 0) = 2^{2i}.$$

Thus the net change in the sum is 0. □

9.3 Encoding Carry-Save Sums

In the context of an iterative multiplication-based algorithm (such as the division and square root algorithms of Chap. 10), it often occurs that the result of a multiplication is used as the multiplier y in the next iteration. In this case, if the Booth encoding of y can be derived directly from the carry-save representation produced by the compression tree, then y need not be computed explicitly and the expensive final step of carry-propagate addition may be avoided.

In this section, we shall assume once again that x is an $(n - 1)$-bit vector, but now the multiplier to be encoded is expressed as a sum

$$y = a + b,$$

where a and b are bit vectors of width $2m - 2$. As a consequence,

$$y \leq (2^{2m-2} - 1) + (2^{2m-2} - 1) = 2^{2m-1} - 2 < 2^{2m-1}.$$

Our objective is to encode y as a sequence of digits $\psi_0, \ldots, \psi_{m-1}$ and derive a modified version of Lemma 9.3 with θ_i replaced by ψ_i. Examining the proofs of Lemmas 9.2 and 9.3, we see that the relevant properties of θ_i are as follows:

(1) $y = \sum_{i=0}^{m-1} 2^{2i}\theta_i$;
(2) $\theta_i \in \{-2, -1, 0, 1, 2\}$ for $i = 0, \ldots, m-1$;
(3) $\theta_{m-1} \geq 0$.

We shall show that the same properties hold for the digits ψ_i, defined as follows.

Definition 9.2 Let $a \in \mathbb{N}$, $b \in \mathbb{N}$, $c \in \mathbb{N}$, and $d \in \mathbb{N}$. For all $i \in \mathbb{N}$, let

$$a_i = a[2i+1:2i]$$

and

$$b_i = b[2i+1:2i],$$

and let γ_i and δ_i be defined recursively as follows: $\gamma_0 = 0$, $\delta_0 = 0$, and for $i > 0$,

$$\gamma_i = a_{i-1}[1] \mid b_{i-1}[1]$$

and

$$\delta_i = (a_{i-1}[0] \,\&\, b_{i-1}[0] \mid a_{i-1}[0] \,\&\, \gamma_{i-1}[0] \mid b_{i-1}[0] \,\&\, \gamma_{i-1}[0]) \\ \&{\sim}(a_{i-1}[1] \,\hat{}\, b_{i-1}[1]).$$

Then for all $i \in \mathbb{N}$,

$$\psi_i(a,b) = a_i + b_i + \gamma_i + \delta_i - 4(\gamma_{i+1} + \delta_{i+1}).$$

Lemma 9.5 *Let $m \in \mathbb{Z}^+$ and $y = a + b$, where a and b are $(2m-2)$-bit vectors. Then*

$$y = \sum_{i=0}^{m-1} 2^{2i}\psi_i(a,b).$$

Proof Let $\psi_i = \psi_i(a,b,c,d)$ and let a_i, b_i, γ_i, and δ_i be as specified in Definition 9.2. We shall prove, by induction, that for $0 \leq k \leq m$,

$$a[2k\text{–}1:0] + b[2k\text{–}1:0] = \sum_{i=0}^{k-1} 2^{2i}\psi_i + 2^k(\gamma_k + \delta_k).$$

Assume that the statement holds for some $k < m$. Then

$$\begin{aligned} a[2k+1:0] + b[2k+1:0] &= 2^k a_k + a[2k\text{–}1:0] + 2^k b_k + b[2k\text{–}1:0] \\ &= 2^k(a_k + b_k) + \sum_{i=0}^{k-1} 2^{2i}\psi_i + 2^k(\gamma_k + \delta_k) \end{aligned}$$

$$= 2^k(a_k + b_k + \gamma_k + \delta_k) + \sum_{i=0}^{k-1} 2^{2i}\psi_i$$

$$= 2^k(\psi_k + 4(\gamma_{k+1} + \delta_{k+1})) + \sum_{i=0}^{k-1} 2^{2i}\psi_i$$

$$= \sum_{i=0}^{k} 2^{2i}\psi_i + 2^{k+1}(\gamma_{k+1} + \delta_{k+1}).$$

Note that $a_{m-1} = b_{m-1} = 0$ and therefore, as a consequence of Definition 9.2, $\gamma_m = \delta_m = 0$. Thus,

$$\begin{aligned} a + b &= a[2m-1:0] + b[2m-1:0] \\ &= \sum_{i=0}^{m-1} 2^{2i}\psi_i + 2^m(\gamma_m + \delta_m) \\ &= \sum_{i=0}^{m-1} 2^{2i}\psi_i. \end{aligned}$$

□

It is not obvious that the ψ_i lie within the prescribed range.

Lemma 9.6 *Let a and b be $(2m-2)$-bit vectors, where $m \in \mathbb{Z}^+$. Then for $i = 0, \ldots, m-1$,*

$$|\psi_i(a, b)| \leq 2.$$

Proof Let $\psi_i = \psi_i(a, b)$ and let a_i, b_i, γ_i, and δ_i be as specified in Definition 9.2. Then

$$\psi_i = a_i + b_i + \gamma_i + \delta_i - 4(\gamma_{i+1} + \delta_{i+1}),$$

where γ_{i+1} and δ_{i+1} are functions of a_i, b_i, and γ_i. Thus, we may express ψ_i as a function of a_i, b_i, γ_i, and δ_i. The inequality may then be trivially verified for each of the $4 \cdot 4 \cdot 2 \cdot 2 = 64$ possible sets of values of these arguments. □

The remaining required property is trivial:

Lemma 9.7 *If a and b be $(2m-2)$-bit vectors, where $m \in \mathbb{Z}^+$, then*

$$\psi_{m-1}(a, b) \geq 0.$$

Proof According to Definition 9.2, $a_{m-1} = b_{m-1} = 0$, $\gamma_m = \delta_m = 0$, and hence $\psi_{m-1} \geq 0$. □

By the same arguments used in the proofs of Lemmas 9.2 and 9.3, we have the following:

Lemma 9.8 *Let $m \in \mathbb{Z}^+$ and $n \in \mathbb{Z}^+$. Let x be an $(n-1)$-bit vector and let $y = a + b$, where a and b are $(2m-2)$-bit vectors. For $i = 0, \ldots, m-1$, let pp_i be as defined in Lemma 9.3 with $\theta_i(y)$ replaced by $\psi_i(a, b)$. Then*

$$\sum_{i=0}^{m-1} pp_i \equiv xy \pmod{2^{n+2m}}.$$

9.4 Statically Encoded Multiplier Arrays

In a practical implementation of the algorithm of Sect. 9.2, although the Booth digits θ_i are not actually computed arithmetically as suggested by the formula of Lemma 9.1, some combinational logic is required to derive an encoding of each θ_i from the corresponding multiplier bits $y[2i+1 : 2i-1]$. If the value of the multiplier is known in advance, i.e., at design time, then these encoded values may be stored instead of the multiplier itself, thereby saving the time and hardware associated with the encoding logic. However, since the range of θ_i consists of 5 values, 3 bits are required for each encoding, and therefore $3m$ bits in total, as compared to $2m$ bits for unencoded multiplier. For a single multiplier, this is a negligible expense, but if the multiplier is to be selected from a array of vectors, then the penalty incurred by such static encoding could be a 50% increase in the size of a large ROM.

In this section, we present an alternative encoding scheme that involves 4 rather than 5 encoded values, which allows 2-bit encodings and thereby eliminates any increase in space incurred by statically encoded arrays. Again we assume that the multiplicand x is an $(n-1)$-bit vector, but the bound on the multiplier is weakened, requiring only that

$$y \leq \sum_{i=0}^{m-1} 2^{2i+1} = \frac{2}{3}(2^{2m} - 1),$$

which implies that y is a $2m$-bit vector. Under this scheme, the coefficients θ_i are replaced with the values ϕ_i defined below. Note that the recursive nature of this definition precludes parallel computation of the m values. Consequently, this technique is not suitable for designs that require dynamic encoding.

The definition of ϕ_i involves a pair of mutually recursive auxiliary functions.

Definition 9.3 For all $y \in \mathbb{N}$ and $i \in \mathbb{N}$,

(a) $\mu_i(y) = y[2i+1:2i] + \chi_i(y)$;

(b) $\chi_i(y) = \begin{cases} 1 \text{ if } i > 0 \text{ and } \mu_{i-1}(y) \geq 3 \\ 0 \text{ otherwise;} \end{cases}$

(c) $\phi_i(y) = \begin{cases} -1 & \text{if } \mu_i[1:0] = 3 \\ \mu_i[1:0] & \text{if } \mu_i[1:0] \neq 3, \end{cases}$ where $\mu_i = \mu_i(y)$.

Thus, ϕ_i is limited to a set of 4 values, $\{-1, 0, 1, 2\}$, and in particular, the second property of θ_i listed in Sect. 9.3 is satisfied. We shall establish the other two properties as well. The proof that $\phi_{m-1} \geq 0$ involves a nontrivial induction.

Lemma 9.9 *Let $y \in \mathbb{N}$ and $m \in \mathbb{N}$. If $y \leq \sum_{i=0}^{m-1} 2^{2i+1}$, then $\chi_m(y)$=0.*

Proof Let $\chi_i = \chi_i(y)$ and $\mu_i = \mu_i(y)$. More generally, we shall prove that for $0 \leq k \leq m$, if $y[2k\text{–}1:0] \leq \sum_{i=0}^{k-1} 2^{2i+1}$, then $\chi_k = 0$. Assuming that this claim holds for some $k < m$ and proceeding by induction, we must show that if $y[2k{+}1:0] \leq \sum_{i=0}^{k} 2^{2i+1}$, then $\chi_{k+1} = 0$.

First, suppose that $y[2k+1:2k] \leq 1$. Then

$$\mu_k = y[2k{+}1:2k] + \chi_k \leq 1 + 1 < 3,$$

and hence $\chi_{k+1} = 0$. Thus, we may assume $y[2k+1:2k] \geq 2$. Since

$$\begin{aligned} 2^{2k} y[2k{+}1:2k] + y[2k\text{–}1:0] &= y[2k{+}1:0] \\ &\leq \sum_{i=0}^{k} 2^{2i+1} \\ &= 2^{2k+1} + \sum_{i=0}^{k-1} 2^{2i+1} \\ &< 2^{2k+1} + 2^{2k} \\ &= 2^{2k} \cdot 3, \end{aligned}$$

we must have $y[2k{+}1:2k] = 2$ and $y[2k\text{–}1:0] \leq \sum_{i=0}^{k-1} 2^{2i+1}$. Now, the inductive hypothesis yields $\chi_k = 0$. Hence,

$$\mu_k = y[2k{+}1:2k] + \chi_k = 2 + 0 < 3,$$

and once again, $\chi_{k+1} = 0$. □

The desired result now follows from Lemma 9.9 and Definitions 9.3.

Corollary 9.10 *If $y \leq \sum_{i=0}^{m-1} 2^{2i+1}$, then $\phi_{m-1} \geq 0$.*

Lemma 9.9 is also needed for the remaining required property of ϕ_i:

Lemma 9.11 *Let $y \in \mathbb{N}$ and $m \in \mathbb{N}$. If $y \le \sum_{i=0}^{m-1} 2^{2i+1}$, then*

$$y = \sum_{i=0}^{m-1} 2^{2i} \phi_i(y).$$

Proof Let $\chi_i = \chi_i(y)$, $\mu_i = \mu_i(y)$, and $\phi_i = \phi_i(y)$. First note that $4\chi_{i+1} + \phi_i = \mu_i$, for if $\mu_i = 3$, then

$$4\chi_{i+1} + \phi_i = 4 - 1 = 3,$$

and in all other cases,

$$4\chi_{i+1} + \phi_i = 4\mu_i[2] + \mu[1:0] = \mu_i.$$

We shall show that for $k = 0, \ldots, m$,

$$y[2k\text{–}1:0] = \sum_{i=0}^{k-1} 2^{2i}\phi_i + 2^{2k}\chi_k.$$

The claim is trivial for $k = 0$. For $0 \le k < m$, by induction, we have

$$\begin{aligned} y[2k{+}1:0] &= 2^{2k} y[2k{+}1:2k] + y[2k\text{–}1:0] \\ &= 2^{2k} y[2k{+}1:2k] + \sum_{i=0}^{k-1} 2^{2i}\phi_i + 2^{2k}\chi_k \\ &= 2^{2k}(y[2k{+}1:2k] + \chi_k) + \sum_{i=0}^{k-1} 2^{2i}\phi_i \\ &= 2^{2k}\mu_k + \sum_{i=0}^{k-1} 2^{2i}\phi_i \\ &= 2^{2k}(4\chi_{k+1} + \phi_k) + \sum_{i=0}^{k-1} 2^{2i}\phi_i \\ &= \sum_{i=0}^{k} 2^{2i}\phi_i + 2^{2(k+1)}\chi_{k+1}. \end{aligned}$$

In particular, substituting $m - 1$ for k, we have

$$y = y[2m\text{–}1:0] = \sum_{i=0}^{m-1} 2^{2i}\phi_i + 2^{2m}\chi_m = \sum_{i=0}^{m-1} 2^{2i}\phi_i.$$

□

Thus, our multiplier y may be statically encoded as a vector of width $2m$,

$$z = \{\mu_{m-1}[1:0], \ldots, \mu_0[1:0]\},$$

from which each ϕ_i may be readily recovered as

$$\phi_i = \begin{cases} -1 & \text{if } z[2i+1:2i] = 3 \\ \mu_i[1:0] & \text{if } z[2i+1:2i] \neq 3. \end{cases}$$

We may now conclude the following result. Note that as a further optimization, the 5-to-1 multiplexer that produces the B_i of Lemma 9.2 is replaced with a 4-to-1 multiplexer.

Lemma 9.12 *Let $m \in \mathbb{Z}^+$ and $n \in \mathbb{Z}^+$. Let x be a bit vector of width $n-1$ and let $y \in \mathbb{N}$ satisfy $y \leq \sum_{i=0}^{m-1} 2^{2i+1}$. For $i = 0, \ldots, m-1$, let pp_i be as defined in Lemma 9.3 with $\theta_i(y)$ replaced by $\phi_i(y)$. Then*

$$\sum_{i=0}^{m-1} pp_i \equiv xy \pmod{2^{n+2m}}.$$

9.5 Radix-8 Booth Encoding

A partition of the multiplier y into slices of three bits instead of two leads to a decomposition

$$y = \sum 2^{3i}\eta_i,$$

where

$$\begin{aligned} \eta_i &= 4\beta_{3i+2} + 2\beta_{3i+1} + \beta_{3i} \\ &= 4(y[3i+1] - y[3i+2]) + 2(y[3i] - y[3i+1]) + (y[3i-1] - y[3i]) \\ &= y[3i-1] + y[3i] + 2y[3i+1] - 4y[3i+2]. \end{aligned}$$

While the number of terms of this sum is only 1/3 (rather than 1/2) of the width of y, the range of digits is now $-4 \leq \eta_i \leq 4$, and hence the value of each term is no longer guaranteed to be a power of 2 in absolute value. Consequently, a multiplier based on this radix-8 scheme generates fewer partial products than a radix-4 multiplier, but the computation of each partial product is more complex. In particular, a partial product corresponding to an encoding $\eta_i = \pm 3$ requires the computation of $3x$, and therefore a full addition.

While radix-4 multiplication is more common, radix-8 may offer an advantage, depending on the timing details of a hardware design. In a typical implementation, the partial products are computed in one clock cycle and the compression tree is executed in the next. If there is sufficient time during the first cycle to perform the addition required for radix-8 encoding (which may be the case, for example, for a low-precision operation), then this scheme is feasible. Since most of the silicon area allocated to a multiplier is associated with the compression tree, the resulting reduction in the number of partial products may represent a significant gain in efficiency.

For the purpose of this analysis, which is otherwise quite similar to that of Sect. 9.2, we shall assume that x and y are bit vectors of widths $n-2$ and $3m-1$, respectively.

Definition 9.4 For $y \in \mathbb{N}$ and $i \in \mathbb{N}$,

$$\eta_i(y) = y[3i{-}1] + y[3i] + 2y[3i{+}1] - 4y[3i{+}2].$$

Lemma 9.13 *Let y be a bit vector of width 2^{3m-1}, where $m \in \mathbb{Z}^+$. Then*

$$y = \sum_{i=0}^{m-1} 2^{3i}\eta_i(y).$$

Proof The proof is essentially the same as that of Lemma 9.1. We shall show by induction that for $0 \leq k \leq m$,

$$y[3k{-}1:0] = \sum_{i=0}^{k-1} 2^{3i}\eta_i + 2^{3k}y[3k{-}1],$$

where $\eta_i = \eta_i(y)$. The claim is trivial for $k = 0$. Assuming that it holds for some $k < m$, we have

$$\begin{aligned}
y[3(k{+}1){-}1:0] &= y[3k{+}2:0] \\
&= 2^{3k}y[3k{+}2:3k] + y[3k{-}1:0] \\
&= 2^{3k}(4y[3k{+}2] + 2y[3k{+}1] + y[3k]) + \sum_{i=0}^{k-1} 2^{3i}\eta_i + 2^{3k}y[3k{-}1] \\
&= 2^{3k}(4y[3k{+}2] + 2y[3k{+}1] + y[3k] + y[3k{-}1]) + \sum_{i=0}^{k-1} 2^{3i}\eta_i \\
&= 2^{3k}(8y[3k{+}2] + \eta_k) + \sum_{i=0}^{k-1} 2^{3i}\eta_i \\
&= \sum_{i=0}^{k} 2^{3i}\eta_i + 2^{3(k+1)}y[3(k+1){-}1],
\end{aligned}$$

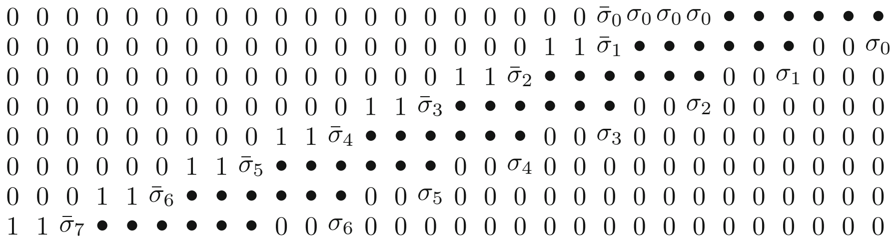


Fig. 9.5 Radix-8 partial product array

which completes the induction. In particular, substituting m for k, we have

$$y = y[3m{-}1:0] = \sum_{i=0}^{m-1} 2^{3i}\eta_i + 2^{3m}y[3m{-}1] = \sum_{i=0}^{m-1} 2^{3i}\eta_i.$$

□

The partial product array, as depicted in Fig. 9.5 for the case $n = 6$, $m = 8$, consists of m bit vectors of width $n + 3m$. Its structure is quite similar to that of the radix-4 array of Lemma 9.3, as its proof of correctness:

Lemma 9.14 *Let x and y be bit vectors of widths $n - 2$ and $3m - 1$, respectively, where $m \in \mathbb{Z}^+$ and $n \in \mathbb{Z}^+$. For $i = 0, \ldots, m - 1$, let $\eta_i = \eta_i(y)$,*

$$\sigma_i = \begin{cases} 0 \text{ if } \eta_i \geq 0 \\ 1 \text{ if } \eta_i < 0, \end{cases}$$

$$\bar{\sigma}_i = 1 - \sigma_i,$$

$$B_i = \begin{cases} \eta_i x & \text{if } \sigma_i = 0 \\ \sim(-\eta_i x)[n{-}1:0] & \text{if } \sigma_i = 1, \end{cases}$$

and

$$pp_i = \begin{cases} \{(3m{-}4)\,'\,0, \bar{\sigma}_0, \sigma_0, \sigma_0, \sigma_0, n\,'\,B_0\} & \text{if } i = 0 \\ \{3(m{-}i{-}1)\,'\,0, 1, 1, \bar{\sigma}_i, n\,'\,B_i, 0, 0, \sigma_{i-1}, 3(i{-}1)\,'\,0\} & \text{if } i > 0. \end{cases}$$

Then

$$\sum_{i=0}^{m-1} pp_i \equiv xy \pmod{2^{n+3m}}.$$

Proof Note that B_i may be expressed as $x\eta_i + (2^n - 1)\sigma_i$. Thus,

$$
\begin{aligned}
pp_0 &= B_0 + 2^n\sigma_0 + 2^{n+1}\sigma_0 + 2^{n+2}\sigma_0 + 2^{n+3}(1-\sigma_0) \\
&= \left(x\eta_0 + (2^n-1)\sigma_0\right) - 2^n\sigma_0 + 2^{n+3} \\
&= x\eta_0 - \sigma_0 + 2^{n+3}
\end{aligned}
$$

and for $i > 0$,

$$
\begin{aligned}
pp_i &= 2^{3(i-1)}\sigma_{i-1} + 2^{3i}B_i + 2^{n+3i}(1-\sigma_i) + 2^{n+3i+1} + 2^{n+3i+2} \\
&= 2^{3(i-1)}\sigma_{i-1} + 2^{3i}\left(x\eta_i + (2^n-1)\sigma_i\right) + 2^{n+3i}(1-\sigma_i) + 2^{n+3i+1} + 2^{n+3i+2} \\
&= 2^{3(i-1)}\sigma_{i-1} + 2^{3i}x\eta_i - 2^{3i}\sigma_i + 2^{n+3(i+1)} - 2^{n+3i}.
\end{aligned}
$$

Combining these expressions and noting that $\sigma_{m-1} = 0$, we have

$$
\begin{aligned}
&\sum_{i=0}^{m-1} pp_i \\
&= \sum_{i=1}^{m-1} 2^{3(i-1)}\sigma_{i-1} + \sum_{i=0}^{m-1} x\eta_i - \sum_{i=0}^{m-1} 2^{3i}\sigma_i + \sum_{i=1}^{m-1}(2^{n+3(i+1)}-2^{n+3i}) + 2^{n+3} \\
&= \sum_{i=0}^{m-2} 2^{3i}\sigma_i + xy - \sum_{i=0}^{m-1} 2^{3i}\sigma_i + 2^{n+3m} \\
&= xy + 2^{n+3m}.
\end{aligned}
$$

□

Chapter 10
SRT Division and Square Root

The simplest and most common approach to computer division is *digit recurrence*, an iterative process whereby at each step, a multiple of the divisor is subtracted from the current remainder and the quotient is updated accordingly by appending a fixed number of bits k, determined by the underlying radix, $r = 2^k$. Thus, quotient convergence is linear, resulting in fairly high latencies of high-precision operations for the most common radices, $r = 2$, 4, and 8.

Since division and square root lend themselves to similar recurrence formulas for the remainder, the same methods are generally applicable to both operations. An important class of algorithms for division and square root are grouped under the name *SRT*, in recognition of the independent contributions of Sweeney[4], Robertson[26], and Tocher[35] in the late 1950s. The common element is a table, indexed by approximations of the divisor or root and the remainder, from which an integer multiplier is extracted on each iteration. SRT dividers are ubiquitous in contemporary microprocessor design and notoriously prone to implementation error. They are, therefore, an important application of formal verification.

Elsewhere [31], we explore the sharing of SRT tables between division and square root as an area-conserving optimization. Here we take a simpler approach using separate tables, following [5]. The results of this chapter are the basis of the floating-point division and square root designs of Chaps. 18 and 19.

10.1 SRT Division

Our objective is to compute an approximation of the quotient $\frac{x}{d}$ of given positive rational numbers x and d. We shall assume that after an appropriate shift,

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_10

$$d \leq x < 2d, \tag{10.1}$$

thereby confining the quotient to the interval $[1, 2)$.

The computation is governed by a fixed power of 2, $r \geq 2$, the *radix* of the operation, which determines the number of bits contributed to the quotient on each iteration. We shall construct a sequence of *quotient digits* $q_j \in \mathbb{Z}$, and the resulting sequence of *partial quotients*,

$$Q_j = \sum_{i=1}^{j} r^{1-i} q_i, \tag{10.2}$$

which converges to $\frac{x}{d}$. We also define the *partial remainders*,

$$R_j = r^{j-1}(x - dQ_j), \tag{10.3}$$

which may be computed by the following recurrence relation:

Lemma 10.1 *$R_0 = \frac{x}{r}$ and for $j \geq 0$, $R_{j+1} = rR_j - q_{j+1}d$.*

Proof The claim is trivial for $j = 0$, and by induction,

$$\begin{aligned} R_{j+1} &= r^j(x - Q_{j+1}d) \\ &= r^j\left(x - (Q_j + r^{-j}q_{j+1}d)\right) \\ &= r^j(x - Q_jd) - q_{j+1}d \\ &= rR_j - q_{j+1}d. \end{aligned}$$

□

The quotient digits are selected from a set of integers $\{-a, \ldots, a\}$, where a is chosen so that the *redundancy factor*

$$\rho = \frac{a}{r-1}$$

satisfies

$$\frac{1}{2} < \rho \leq 1. \tag{10.4}$$

The *minimally redundant* case $a = \frac{r}{2}$ minimizes the number of multiples of d that must be computed, while the *maximally redundant* case $a = r - 1$ provides greater flexibility in the selection of digits.

The digits are selected with the goal of preserving the invariant

$$|R_j| \leq \rho d, \tag{10.5}$$

or equivalently,

$$\left|\frac{x}{d} - Q_j\right| \leq \rho r^{1-j}. \tag{10.6}$$

This choice of bound is motivated by the observation that if

$$\frac{x}{d} = \lim_{j \to \infty} Q_j = \sum_{i=1}^{\infty} r^{1-i} q_i,$$

then

$$\left|\frac{x}{d} - Q_j\right| = \left|\sum_{i=j+1}^{\infty} r^{1-i} q_i\right| \leq a \sum_{i=j+1}^{\infty} r^{1-i} = \frac{ar^{1-j}}{r-1} = \rho r^{1-j}.$$

Thus, (10.6) is equivalent to convergence.

The invariant holds trivially for $j = 0$:

Lemma 10.2 $R_0 \leq \rho d$.

Proof If $r = 2$, then $a = 1$, $\rho = 1$, and

$$R_0 = \frac{x}{2} < d = \rho d.$$

Otherwise, $r \geq 4$ and

$$R_0 = \frac{x}{r} < \frac{2d}{r} \leq \frac{d}{2} < \rho d.$$

□

For $k \in \{-a, \ldots, a\}$, the *selection interval* $[L_k(d), U_k(d)]$, defined by

$$U_k(d) = (k + \rho)d$$

and

$$L_k(d) = (k - \rho)d,$$

is so named because if the shifted partial remainder rR_j lies in this interval, then the invariant (10.5) may be preserved by choosing $q_{j+1} = k$:

Lemma 10.3 *If $L_k(d) \leq rR_j \leq U_k(d)$ and $q_{j+1} = k$, then $|R_{j+1}| \leq \rho d$.*

Proof By Lemma 10.1,

$$\begin{aligned} L_k(d) \leq rR_j \leq U_k(d) &\Rightarrow (k-\rho)d \leq rR_j \leq (k+\rho)d \\ &\Rightarrow -\rho d \leq rR_j - kd \leq \rho d \\ &\Rightarrow -\rho d \leq R_{j+1} \leq \rho d. \end{aligned}$$

□

Thus, the existence of a root digit q_{j+1} that preserves (10.5) is guaranteed if the selection intervals cover the entire range of rR_j, i.e.,

$$[-r\rho d, r\rho d] \subseteq \bigcup_{k=-a}^{a} [L_k(j), U_k(j)] \,.$$

This is ensured by the following two lemmas. Note that the proof of the first accounts for the bounds (10.4) imposed on ρ:

Lemma 10.4 *For all $k \in \mathbb{Z}$,*

(a) $L_k(d) < L_{k+1}(d) < U_k(d) < U_{k+1}(d)$;
(b) $U_k(d) \leq L_{k+2}(d)$.

Proof

(a) $L_{k+1}(d) - L_k(d) = U_{k+1}(d) - U_k(d) = d > 0$ and $U_k(d) - L_{k+1}(d) = (2\rho - 1)d > 0$.
(b) $L_{k+2}(d) - U_k(d) = 2(1-\rho)d \geq 0$. □

Lemma 10.5 *$U_a(d) = r\rho d$ and $L_{-a}(d) = -r\rho d$.*

Proof First note that

$$a + \rho = \frac{a(r-1) + a}{r-1} = \frac{ar}{r-1} = r\rho.$$

Thus, $U_a(d) = (a+\rho)d = r\rho d$ and $L_{-a}(d) = (-a-\rho)d = -r\rho d$. □

In practice, the selection of a quotient digit that satisfies the hypothesis of Lemma 10.3 is achieved by means of a set of *comparison constants* $m_k(d)$, $-a < k \leq a$, which satisfy $L_k(d) < m_k(d) \leq U_{k-1}(d)$. Note that if $-a < k < a$, then this condition, along with Lemma 10.4 (b), implies that $m_k(d) \leq U_{k-1}(d) \leq L_{k+1}(d) < m_{k+1}(d)$, i.e., $m_k(d)$ is a strictly increasing function of k. As a matter of convenience, we also define $m_{-a}(d) = -\infty$. If q_{j+1} is selected as the largest k for which $m_k(d) \leq rR_j$, then the required bound on $|R_{j+1}|$ follows from Lemmas 10.3 and 10.5.

Typically, however, the partial remainder R_j is represented in a redundant form, i.e., as a sum or difference of two vectors, and therefore, such direct comparisons are not possible. Instead, the comparisons are based on an approximation A_j of rR_j, produced by a narrow adder applied to the leading bits of the component vectors of R_j. Such an implementation is based on an integer parameter t, the number of fractional bits of A_j, which determines the width of the adder and the accuracy of the approximation.

In the SRT divider of Chap. 18, for example, rR_j is represented in *sign-digit* form, i.e., as a difference of two bit vectors, $rR_j = P_j - N_j$, which are truncated to t fractional bits and subtracted to produce an approximation

$$A_j = P_j^{(t)} - N_j^{(t)},$$

which satisfies

$$|A_j - rR_j| < 2^{-t}.$$

In a design that represents the shifted remainder as a sum rather than a difference, $rR_j = C_j + S_j$, the same error range may be effected by decrementing the truncated sum, resulting in

$$A_j = C_j^{(t)} + S_j^{(t)} - 2^{-t}.$$

In order to simplify the comparison with A_j, the constants $m_k(d)$ should be selected to have as few fractional bits as possible. We need only assume, however, that $m_k(d)$ has at most t fractional bits.

Lemma 10.6 *Let $t \in \mathbb{N}$. Let $m_{-a}(d) = -\infty$ and for $-a < k \leq a$, let $m_k(d) \in \mathbb{Q}$ such that $2^t m_k(d) \in \mathbb{Z}$ and*

$$L_k(d) + 2^{-t} \leq m_k(d) \leq U_{k-1}(d).$$

Let $j \in \mathbb{N}$ and assume that $|R_j| \leq \rho d$. Let $A_j \in \mathbb{Q}$ such that $2^t A_j \in \mathbb{Z}$ and

$$|A_j - rR_j| < 2^{-t}.$$

If q_{j+1} is the greatest $k \in \{-a, \ldots, a\}$ such that $m_k(d) \leq A_j$, then $|R_{j+1}| \leq \rho d$.

Proof Let $q = q_{j+1}$. We shall show that $L_q(d) \leq rR_j \leq U_q(d)$ and invoke Lemma 10.3. Since Lemma 10.5 ensures that $L_{-a}(d) \leq rR_j \leq U_a(d)$, the required bounds are reduced to

$$-a < q \leq a \Rightarrow rR_j \geq L_q(d)$$

and

$$-a \leq q < a \Rightarrow rR_j \leq U_q(d).$$

But according to hypothesis, if $-a < q \leq a$, then

$$L_q(d) \leq m_q(d) - 2^{-t} \leq A_j - 2^{-t} < rR_j$$

and if $-a \leq q < a$, then $m_{q+1}(d) > A_j$, which implies $m_{q+1}(d) \geq A_j + 2^{-t}$, and hence

$$U_q(d) \geq m_{q+1}(d) \geq A_j + 2^{-t} > rR_j.$$

□

The parameter t must be large enough to ensure the existence of comparison constants $m_k(d)$ that satisfy the hypothesis of Lemma 10.6, but should be as small as possible in order to minimize the width of the adder that generates A_j. The number of fractional bits of these constants is at most t (as required by the lemma) and should be further reduced if possible in order to simplify the comparisons.

10.2 Minimally Redundant Radix-4 Division

As an example, we consider the case $r = 4$, $a = 2$ and $\rho = \frac{2}{3}$. This is a particularly common instance of SRT division because multiplication of the divisor by each element of the digit set $\{-2, -1, 0, 1, 2\}$ may be performed as a simple shift of d or its complement. A common technique for limiting the size of the table of comparison constants is *prescaling*, whereby x and d are multiplied by the same factor (thus preserving the quotient) in order to confine d to a small neighborhood of 1. This allows the same four comparison constants m_k, $-1 \leq k \leq 2$, to be used for all values of d. In Chap. 18, we present an implementation based on an efficient prescaling procedure that results in the bounds

$$\frac{63}{64} \leq d \leq \frac{9}{8}. \tag{10.7}$$

According to Lemma 10.6, we must select a value of t for which there exist m_k such that $2^t m_k \in \mathbb{Z}$ and

$$L_k(d) + 2^{-t} \leq m_k \leq U_{k-1}(d)$$

for all d satisfying (10.7). Let

$$\overline{L}_k = \max\left(L_k\left(\frac{63}{64}\right), L_k\left(\frac{9}{8}\right)\right)$$

and

$$\underline{U}_k = \min\left(U_k\left(\frac{63}{64}\right), U_k\left(\frac{9}{8}\right)\right).$$

m_2	m_1	m_0	m_{-1}	m_{-2}
$\frac{13}{8}$	$\frac{4}{8}$	$-\frac{3}{8}$	$-\frac{12}{8}$	$-\infty$

Fig. 10.1 Comparison constants m_k

Since L_k and U_k are linear functions of d, it follows from (10.7) that

$$L_k(d) \leq \overline{L}_k$$

and

$$U_k(d) \geq \underline{U}_k.$$

Our requirement, therefore, is reduced to

$$\overline{L}_k + 2^{-t} \leq m_k \leq \underline{U}_{k-1}, \tag{10.8}$$

or

$$2^t\overline{L}_k + 1 \leq 2^t m_k \leq 2^t \underline{U}_{k-1},$$

where $2^t m_k \in \mathbb{Z}$. Thus, we must select t to be large enough that there exists an integer in the interval $[2^t\overline{L}_k + 1, 2^t\underline{U}_{k-1}]$, i.e., that

$$\lceil 2^t\overline{L}_k + 1\rceil \leq \lfloor 2^t\underline{U}_{k-1}\rfloor,$$

for $-1 \leq k \leq 2$. It is easily verified that $\overline{L}_k < \underline{U}_{k-1}$ for each k, which implies the satisfiability of this constraint, and that the smallest solution is $t = 3$. Moreover, (10.8) is satisfied for this value of t by the constants m_k displayed in Fig. 10.1. (No compliant set of constants exists with fewer than 3 fractional bits.) Thus, we have the following instantiation of Lemma 10.6.which is the basis of the inductive proof of convergence for the module of Chap. 18:

Lemma 10.7 *Let $r = 4$ and $a = 2$ and let m_k be as listed in Fig. 10.1 for $-2 \leq k \leq 2$. Assume that*

$$\frac{63}{64} \leq d \leq \frac{9}{8}.$$

Let $j \geq 0$ and assume that the following conditions hold:

(a) $|R_j| \leq \frac{2}{3}d$;
(b) $A_j \in \mathbb{Q}$ satisfies $8A_j \in \mathbb{Z}$ and $|A_j - 4R_j| < \frac{1}{8}$;
(c) q_{j+1} is the greatest $k \in \{-2, \ldots, 2\}$ such that $m_k \leq A_j$.

Then $|R_{j+1}| \leq \frac{2}{3}d$.

10.3 Minimally Redundant Radix-8 Division

In a variant of the processor in which the above radix-4 divider is implemented, the radix is increased to 8, thereby providing an extra quotient bit per iteration at the expense of increased complexity. This version is also minimally redundant, with $a = 4$ and $\rho = \frac{4}{7}$.

Prescaling of the divisor in this design would require a multiplier with 7 partial products (as opposed to 3 in the radix-4 case), which was found to be infeasible with respect to timing. Instead, we merely assume that

$$\frac{1}{2} \leq d < 1. \tag{10.9}$$

Consequently, the comparison constants are dependent on d. Specifically, they are based on a partition of the range of d into 64 subintervals. Let

$$i = \left\lfloor 128\left(d - \frac{1}{2}\right)\right\rfloor,$$

so that $0 \leq i < 64$ and

$$\frac{1}{2} + \frac{i}{128} \leq d < \frac{1}{2} + \frac{i+1}{128}. \tag{10.10}$$

The comparison constants are determined by i; that is, we define a set of constants $m_k(i)$ for each of these intervals.

For $-4 \leq k \leq 4$, let

$$\overline{L}_k(i) = \max\left(L_k\left(\frac{1}{2} + \frac{i}{128}\right), L_k\left(\frac{1}{2} + \frac{i+1}{128}\right)\right)$$

and

$$\underline{U}_k(i) = \min\left(U_k\left(\frac{1}{2} + \frac{i}{128}\right), U_k\left(\frac{1}{2} + \frac{i+1}{128}\right)\right).$$

Once again, $L_k(d)$ and $U_k(d)$ are linear functions of d, and hence

$$L_k(d) \leq \overline{L}_k(i) \text{ and } U_k(d) \geq \underline{U}_k(i). \tag{10.11}$$

Thus, we seek $t \in \mathbb{N}$ and constants $m_k(i)$, where $-3 \leq k \leq 4$ and $0 \leq i < 64$, such that $2^t m_k(i) \in \mathbb{Z}$ and

$$\overline{L}_k(i) + 2^t \leq m_k(i) \leq \underline{U}_{k-1}(i). \tag{10.12}$$

By the same reasoning as used in the radix-4 case, a solution exists if and only if

$$\lceil 2^t \overline{L}_k(i) + 1 \rceil \le \lfloor \underline{U}_{k-1}(i) \rfloor$$

for all i. By direct computation, the minimal value for which this condition holds is $t = 6$.

According to Lemma 10.5 (d), each constant satisfies

$$|m_k(i)| < r\rho d < 8 \cdot \frac{4}{7} < 5$$

and may therefore be represented with 10 bits, 4 integer and 6 fractional. It follows that we can construct a table of size $64 \times 8 \times 10 = 5120$ bits, from which the appropriate 8 10-bit constants may be extracted according to the leading 6 bits of d.

But closer inspection reveals that the constants may be chosen to be independent of the least significant bit of i with very few exceptions, an observation that may be exploited effectively to reduce the table size by half. We shall define $m_k(i)$ according to Fig. 10.2. Note that for a given value of i, the constants $m_k(i)$ are derived by a table access based on the 5-bit value $\lfloor i/2 \rfloor$, and require possible adjustment according to the 6th bit only in the four cases $i < 4$.

Since these constants are readily shown to satisfy (10.12) for $t = 6$, we have the following consequence of Lemma 10.6:

Lemma 10.8 *Let $r = 8$ and $a = 4$ and let $m_k(i)$ be as listed in Fig. 10.2 for $-4 \le k \le 4$, where*

$$i = \left\lfloor 128 \left(d - \frac{1}{2} \right) \right\rfloor .$$

Let $j \ge 0$ and assume that the following conditions hold:

(a) $|R_j| \le \frac{4}{7}d$;
(b) $A_j \in \mathbb{Q}$ satisfies $64A_j \in \mathbb{Z}$ and $|A_j - 8R_j| < \frac{1}{64}$;
(c) q_{j+1} is the greatest $k \in \{-4, \ldots, 4\}$ such that $m_k(i) \le A_j$.

Then $|R_{j+1}| \le \frac{4}{7}d$.

10.4 SRT Square Root

Let r be a fixed power of 2. Given a positive rational number x in the range $\frac{1}{4} \le x < 1$, our objective is to construct a sequence of *root digits* $q_j \in \mathbb{Z}$, and a corresponding sequence of *partial roots*

$$Q_j = 1 + \sum_{i=1}^{j} r^{-i} q_i, \tag{10.13}$$

which converge to $\sqrt{x} \in [\frac{1}{2}, 1)$. Note that for all $j \in \mathbb{N}$, $r^j Q_j \in \mathbb{Z}$.

$\lfloor i/2 \rfloor$	m_4	m_3	m_2	m_1	m_0	m_{-1}	m_{-2}	m_{-3}	m_{-4}
0	115/64*	82/64	50/64	16/64	−16/64	−48/64	−81/64	-112/64*	$-\infty$
1	118/64*	84/64	50/64	16/64	−16/64	−50/64	−83/64	-116/64*	$-\infty$
2	121/64	86/64	52/64	16/64	−16/64	−52/64	−86/64	-120/64	$-\infty$
3	125/64	90/64	54/64	18/64	−18/64	−54/64	−88/64	-124/64	$-\infty$
4	128/64	92/64	54/64	18/64	−18/64	−54/64	−90/64	-127/64	$-\infty$
5	132/64	94/64	56/64	18/64	−18/64	−56/64	−94/64	-131/64	$-\infty$
6	135/64	96/64	58/64	18/64	−18/64	−58/64	−96/64	-134/64	$-\infty$
7	139/64	100/64	60/64	20/64	−20/64	−60/64	-98/64	-138/64	$-\infty$
8	142/64	102/64	60/64	20/64	−20/64	−60/64	-100/64	-141/64	$-\infty$
9	146/64	104/64	62/64	20/64	−20/64	−62/64	-104/64	-144/64	$-\infty$
10	150/64	106/64	64/64	20/64	−20/64	−64/64	-106/64	-148/64	$-\infty$
11	152/64	108/64	64/64	20/64	−20/64	−64/64	-108/64	-152/64	$-\infty$
12	156/64	112/64	66/64	22/64	−22/64	−66/64	-112/64	-156/64	$-\infty$
13	160/64	114/64	68/64	22/64	−22/64	−68/64	-114/64	-158/64	$-\infty$
14	164/64	116/64	70/64	24/64	−24/64	−70/64	-116/64	-162/64	$-\infty$
15	166/64	118/64	70/64	24/64	−24/64	−70/64	-118/64	-166/64	$-\infty$
16	170/64	120/64	72/64	24/64	−24/64	−72/64	-120/64	-170/64	$-\infty$
17	173/64	124/64	73/64	24/64	−24/64	−72/64	-124/64	-172/64	$-\infty$
18	176/64	126/64	76/64	24/64	−24/64	−76/64	-124/64	-176/64	$-\infty$
19	180/64	128/64	76/64	24/64	−24/64	−76/64	-128/64	-180/64	$-\infty$
20	184/64	132/64	78/64	24/64	−24/64	−78/64	-132/64	-184/64	$-\infty$
21	188/64	134/64	80/64	28/64	−28/64	−80/64	-134/64	-188/64	$-\infty$
22	190/64	136/64	82/64	28/64	−28/64	−82/64	-136/64	-190/64	$-\infty$
23	194/64	138/64	82/64	28/64	−28/64	−82/64	-138/64	-194/64	$-\infty$
24	198/64	140/64	84/64	28/64	−28/64	−84/64	-140/64	-198/64	$-\infty$
25	200/64	142/64	84/64	28/64	−28/64	−84/64	-142/64	-200/64	$-\infty$
26	204/64	146/64	86/64	28/64	−28/64	−86/64	-146/64	-204/64	$-\infty$
27	208/64	148/64	88/64	28/64	−28/64	−88/64	-148/64	-208/64	$-\infty$
28	212/64	152/64	90/64	28/64	−28/64	−90/64	-152/64	-212/64	$-\infty$
29	214/64	152/64	90/64	28/64	−28/64	−90/64	-152/64	-214/64	$-\infty$
30	218/64	154/64	94/64	28/64	−28/64	−94/64	-154/64	-218/64	$-\infty$
31	222/64	158/64	94/64	32/64	−32/64	−94/64	-158/64	-222/64	$-\infty$

* Exceptions: $m_4(0) = \frac{113}{64}$, $m_{-3}(1) = -\frac{114}{64}$, $m_4(2) = \frac{117}{64}$, $m_{-3}(3) = -\frac{117}{64}$

Fig. 10.2 Comparison constants $m_k(i)$, where $i = \left\lfloor 128\left(d - \frac{1}{2}\right)\right\rfloor$

The digits are selected from a set $\{-a, \ldots, a\}$, where a is again chosen so that the redundancy factor $\rho = a/(r-1)$ satisfies (10.4). We define the *partial remainders*,

$$R_j = r^j(x - Q_j^2), \tag{10.14}$$

which are computed by the following recurrence relation:

Lemma 10.9 $R_0 = x - 1$ *and for* $j \geq 0$,

$$R_{j+1} = rR_j - q_{j+1}(2Q_j + r^{-(j+1)}q_{j+1}).$$

Proof The claim is trivial for $j = 0$, and by induction,

$$\begin{aligned}
R_{j+1} &= r^{j+1}(x - Q_{j+1}^2) \\
&= r^{j+1}\left(x - (Q_j + r^{-(j+1)}q_{j+1})^2\right) \\
&= r^{j+1}\left(x - (Q_j^2 + 2Q_j r^{-(j+1)}q_{j+1} + r^{-2(j+1)}q_{j+1}^2)\right) \\
&= r^{j+1}(x - Q_j^2) - r^{j+1}(2Q_j r^{-(j+1)}q_{j+1} + r^{-2j}q_{j+1}^2) \\
&= rR_j - q_{j+1}(2Q_j + r^{-(j+1)}q_{j+1}).
\end{aligned}$$

□

For $j \geq 0$, let

$$\underline{B}(j) = -2\rho Q_j + \rho^2 r^{-j}$$

and

$$\overline{B}(j) = 2\rho Q_j + \rho^2 r^{-j}.$$

The root digits will be selected with the goal of preserving the invariant

$$\underline{B}(j) \leq R_j \leq \overline{B}(j). \tag{10.15}$$

To motivate this choice of bounds, note that if

$$\sqrt{x} = \lim_{j\to\infty} Q_j = \lim_{j\to\infty}\left(1 + \sum_{i=1}^{j} r^{-i}q_i\right) = 1 + \sum_{i=1}^{\infty} r^{-i}q_i,$$

then

$$|\sqrt{x} - Q_j| = \left|\sum_{i=j+1}^{\infty} r^{-i}q_i\right| \leq a \sum_{i=j+1}^{\infty} r^{-i} = \frac{ar^{-j}}{r-1} = \rho r^{-j},$$

or

$$Q_j - \rho r^{-j} \leq \sqrt{x} \leq Q_j + \rho r^{-j}.$$

Since

$$Q_j = 1 + \sum_{i=1}^{j} r^{-i} q_i \geq 1 - \sum_{i=1}^{j} r^{-i} a = 1 - \frac{a(1 - r^{-j})}{r - 1} = 1 - \rho + \rho r^{-j} \geq \rho r^{-j},$$

this may be expressed in rational terms as

$$(Q_j - \rho r^{-j})^2 \leq x \leq (Q_j + \rho r^{-j})^2.$$

Thus, (10.15) is equivalent to convergence:

Lemma 10.10 *For $j \geq 0$,*

$$(Q_j - \rho r^{-j})^2 \leq x \leq (Q_j + \rho r^{-j})^2 \Leftrightarrow \underline{B}(j) \leq R_j \leq \overline{B}(j).$$

Proof

$$\begin{aligned}
&(Q_j - \rho r^{-j})^2 \leq x \leq (Q_j + \rho r^{-j})^2 \\
&\Leftrightarrow Q_j^2 - 2Q_j \rho r^{-j} + \rho^2 r^{-2j} \leq x \leq Q_j^2 + 2Q_j \rho r^{-j} + \rho^2 r^{-2j} \\
&\Leftrightarrow -2\rho Q_j + \rho^2 r^{-j} \leq r^j (x - Q_j^2) \leq 2\rho Q_j + \rho^2 r^{-j} \\
&\Leftrightarrow \underline{B}(j) \leq R_j \leq \overline{B}(j).
\end{aligned}$$

□

Note that the invariant holds trivially for $j = 0$:

Lemma 10.11 $\underline{B}(0) \leq R_0 \leq \overline{B}(0)$.

Proof Since $R_0 = x - 1$ and $\frac{1}{4} \leq x < 1$, we have $-\frac{3}{4} \leq R_0 < 0$,

$$\underline{B}(0) = -2\rho Q_0 + \rho^2 = -2\rho + \rho^2 = (\rho - 1)^2 - 1 \leq \left(\frac{1}{2} - 1\right)^2 - 1 = -\frac{3}{4} \leq R_0,$$

and

$$\overline{B}(0) = 2\rho Q_0 + \rho^2 = 2\rho + \rho^2 \geq 2 \cdot \frac{1}{2} + \frac{1}{2}^2 = \frac{5}{4} > R_0.$$

□

For $j \geq 0$ and $k \in \{-a, \ldots, a\}$, the *selection interval* $[L_k(j), U_k(j)]$, defined by

$$U_k(j) = 2(k+\rho)Q_j + (k+\rho)^2 r^{-(j+1)} \tag{10.16}$$

and

$$L_k(j) = 2(k-\rho)Q_j + (k-\rho)^2 r^{-(j+1)}, \tag{10.17}$$

is so named because if the shifted partial remainder rR_j lies in this interval, then the invariant (10.15) may be ensured by choosing $q_{j+1} = k$:

Lemma 10.12 *Let $j \geq 0$ and $-a \leq k \leq a$. If $L_k(j) \leq rR_j \leq U_k(j)$ and $q_{j+1} = k$, then*

$$\underline{B}(j+1) \leq R_{j+1} \leq \overline{B}(j+1).$$

Proof By hypothesis and Lemma 10.9,

$$\begin{aligned}
R_{j+1} &= rR_j - 2Q_jk - r^{-(j+1)}k^2 \\
&\leq U_k(j) - 2Q_jk - r^{-(j+1)}k^2 \\
&= 2(k+\rho)Q_j + (k+\rho)^2 r^{-(j+1)} - 2Q_jk - r^{-(j+1)}k^2 \\
&= 2\rho Q_j + (2k\rho + \rho^2)r^{-(j+1)} \\
&= 2\rho(Q_{j+1} - r^{-(j+1)}k) + (2k\rho + \rho^2)r^{-(j+1)} \\
&= 2\rho Q_{j+1} + \rho^2 r^{-(j+1)} \\
&= \overline{B}(j+1),
\end{aligned}$$

and similarly,

$$\begin{aligned}
R_{j+1} &\geq L_k(j) - 2Q_jk - r^{-(j+1)}k^2 \\
&= 2(k-\rho)Q_j + (k-\rho)^2 r^{-(j+1)} - 2Q_jk - r^{-(j+1)}k^2 \\
&= -2\rho Q_j + (-2k\rho + \rho^2)r^{-(j+1)} \\
&= -2\rho(Q_{j+1} - r^{-(j+1)}k) + (-2k\rho + \rho^2)r^{-(j+1)} \\
&= -2\rho Q_{j+1} + \rho^2 r^{-(j+1)} \\
&= \underline{B}(j+1).
\end{aligned}$$

□

Thus, the existence of a root digit q_{j+1} that preserves (10.15) is guaranteed if the selection intervals cover the entire range of rR_j, i.e.,

$$\left[r\underline{B}(j), r\overline{B}(j)\right] \subseteq \bigcup_{k=-a}^{a} [L_k(j), U_k(j)]\,.$$

Unlike the corresponding intervals for division, the overlapping of successive selection intervals for the square root depends on the choice of parameters, but the following important property holds in the general case.

Lemma 10.13 *For $j \geq 0$, $U_a(j) = r\overline{B}(j)$ and $L_{-a}(j) = r\underline{B}(j)$.*

Proof As noted in the proof of Lemma 10.5, $a + \rho = r\rho$. Thus,

$$U_a(j) = 2(a+\rho)Q_j + (a+\rho)^2 r^{-(j+1)} = 2r\rho Q_J + r\rho^2 r^{-j} = r\overline{B}(j)$$

and

$$L_{-a}(j) = 2(-a-\rho)Q_j + (-a-\rho)^2 r^{-(j+1)} = -2r\rho Q_J + r\rho^2 r^{-j} = r\underline{B}(j).$$

□

We have the following analog of Lemma 10.6:

Lemma 10.14 *Let $j \in \mathbb{N}$, $t \in \mathbb{N}$, and $A_j \in \mathbb{Q}$. Let $m_{-a}(j) = -\infty$ and for $-a < k \leq a$, let $m_k(j) \in \mathbb{Q}$ such that*

$$2^t m_k(j) \in \mathbb{Z}, \tag{10.18}$$

$$A_j < m_k(j) \Rightarrow rR_j < m_k(j), \tag{10.19}$$

and

$$A_j \geq m_k(j) \Rightarrow rR_j > m_k(j) - 2^{-t}. \tag{10.20}$$

Let $q = q_{j+1}$. Assume that q is the greatest $k \in \{-a, \ldots, a\}$ such that $m_k(j) \leq A_j$, and that

$$q \neq a \Rightarrow m_{q+1}(j) \leq U_q(j) \tag{10.21}$$

and

$$q \neq -a \Rightarrow L_q(j) + 2^{-t} \leq m_q(j). \tag{10.22}$$

If $\underline{B}(j) \leq R_j \leq \overline{B}(j)$, then $\underline{B}(j+1) \leq R_{j+1} \leq \overline{B}(j+1)$.

Proof We shall show that $L_q(j) \leq rR_j \leq U_q(j)$ and invoke Lemma 10.12. Since Lemma 10.13 ensures that $L_{-a}(j) \leq rR_j \leq U_a(j)$, the required bounds are reduced to

$$q \neq a \Rightarrow rR_j \leq U_q(j)$$

and

$$q \neq -a \Rightarrow rR_j \geq L_q(j).$$

But according to hypothesis, if $q \neq a$, then $A_j < m_{q+1}(j)$, which implies

$$rR_j < m_{q+1}(j) \leq U_q(j),$$

and if $q \neq -a$, then $A_j \geq m_q(j)$, which implies

$$rR_j > m_q(j) - 2^{-t} \geq (L_q(j) + 2^{-t}) - 2^{-t} = L_q(j).$$

□

We note that the hypothesis of Lemma 10.14 is weaker than that of Lemma 10.6 in two respects:

(1) The explicit assumption that the parameter t represents the number of fractional bits of the approximation A_j, i.e., that

$$2^t A_j \in \mathbb{Z} \tag{10.23}$$

and

$$|A_j - rR_j| < 2^{-t}, \tag{10.24}$$

is replaced with (10.19) and (10.20). The reason for this is that we find that in practice (see Lemma 19.4), (10.23) generally holds but (10.24) does not. Note that in the context of (10.18) and (10.23), (10.19) and (10.20) together constitute a weakening of (10.24).

As in the case of division, t should be as small as possible in order to minimize the width of the adder required to produce A_j, and the number of fractional bits of $m_k(j)$ should be further reduced if possible in order to simplify the comparisons.

(2) Instead of the general assumption that

$$L_k(j) + 2^{-t} \leq m_k(j) \leq U_k(j) \tag{10.25}$$

for all k, we have the special cases (10.21) and (10.22). Note that the selection interval for $m_k(j)$ now depends on j, and in particular, on Q_j. We shall find that

in practice, (10.25) holds for all k for sufficiently large j (after Q_j stabilizes), but the first several iterations must be handled separately. (See the proof of Lemma 10.15.)

10.5 Minimally Redundant Radix-4 Square Root

As an illustration, we once again consider the case $r = 4$, $a = 2$, $\rho = \frac{2}{3}$, which is the basis of the design of Chap. 19. The problem of parameter optimization for the square root is not as straightforward as for division, but it has been determined experimentally that for this case, the required accuracy of the approximation A_j is given by $t = 5$, with only 3 fractional bits needed for the comparison constants.

Since prescaling of the radicand is not feasible, the constants must depend on the partial root Q_j. Given $j \geq 0$, let $j' = \min(j, 2)$ and let i be the integer defined by $Q_{j'} = \frac{1}{2} + \frac{i}{16}$. We shall show that each partial root satisfies the invariant $\frac{1}{2} \leq Q_j \leq 1$, so that $0 \leq i \leq 8$. The constants, which are displayed in Fig. 10.3, depend primarily on i and are therefore denoted as $m_k(i, j)$. In fact, as noted in the figure, $m_k(i, j)$ is determined solely by i with the three exceptions $m_{-1}(0, j)$, $m_2(1, j)$, and $m_{-1}(8, j)$. Also note that for fixed i and j,

$$m_{-2}(i, j) < m_{-1}(i, j) < m_0(i, j) < m_1(i, j) < m_2(i, j).$$

In Chap. 19, we shall prove that the implementation satisfies the invariant (10.15). The proof will also establish the required bounds on the partial roots. The following result will be the basis of the induction. Note that this induction is based on a slightly different scheme from that used for division: each of Lemmas 10.7 and 10.8 derives the conclusion that a set of properties hold for an index $j + 1$ from the assumption that they hold for j, whereas the proof below requires the explicit assumption that the desired properties hold for all indices between 0 and j.

Lemma 10.15 *Let $r = 4$ and $a = 2$ and let $m_k(i, j)$ be as specified in Fig. 10.3. Given $j \geq 0$, suppose that the following conditions hold for all ℓ, $0 \leq \ell \leq j$:*

(a) $\frac{1}{2} \leq Q_\ell \leq 1$*;*

(b) $\underline{B}(\ell) \leq R_\ell \leq \overline{B}(\ell)$*;*

(c) Let $\ell' = \min(\ell, 2)$ *and* $i = 16\left(Q_{\ell'} - \frac{1}{2}\right)$*. For all* $k \in \{-1, \ldots, 2\}$*,* $A_\ell \in \mathbb{Q}$ *satisfies*

$$A_\ell < m_k(i, \ell) \Rightarrow 4R_\ell < m_k(i, \ell)$$

and

$$A_\ell \geq m_k(i, \ell) \Rightarrow 4R_\ell > m_k(i, \ell) - \frac{1}{32},$$

and $q_{\ell+1}$ *is the greatest* $k \in \{-2, \ldots, 2\}$ *such that* $m_k(i, \ell) \leq A_\ell$*.*

i	m_2	m_1	m_0	m_{-1}	m_{-2}
0	$\frac{12}{8}$	$\frac{4}{8}$	$-\frac{4}{8}$	$-\frac{12}{8}^*$	$-\infty$
1	$\frac{13}{8}^*$	$\frac{4}{8}$	$-\frac{4}{8}$	$-\frac{13}{8}$	$-\infty$
2	$\frac{15}{8}$	$\frac{4}{8}$	$-\frac{4}{8}$	$-\frac{15}{8}$	$-\infty$
3	$\frac{16}{8}$	$\frac{6}{8}$	$-\frac{6}{8}$	$-\frac{16}{8}$	$-\infty$
4	$\frac{18}{8}$	$\frac{6}{8}$	$-\frac{6}{8}$	$-\frac{18}{8}$	$-\infty$
5	$\frac{20}{8}$	$\frac{8}{8}$	$-\frac{6}{8}$	$-\frac{20}{8}$	$-\infty$
6	$\frac{20}{8}$	$\frac{8}{8}$	$-\frac{8}{8}$	$-\frac{20}{8}$	$-\infty$
7	$\frac{22}{8}$	$\frac{8}{8}$	$-\frac{8}{8}$	$-\frac{22}{8}$	$-\infty$
8	$\frac{24}{8}$	$\frac{8}{8}$	$-\frac{8}{8}$	$-\frac{24}{8}^*$	$-\infty$

* Exceptions: $m_{-1}(0,1) = -\frac{11}{8}$, $m_2(1,2) = \frac{15}{8}$, $m_{-1}(8,0) = -\frac{20}{8}$

Fig. 10.3 Comparison constants $m_k(i, j)$

Then (a) and (b) also hold for $\ell = j + 1$.

Proof

(a) If $Q_j = \frac{1}{2}$, then

$$R_j = 4^j \left(x - \frac{1}{4} \right) \geq 0.$$

It follows that $q_{j+1} \geq 0$, for otherwise

$$4R_j < A_j + \frac{1}{32} < m_0(i, j) + \frac{1}{32} < 0.$$

Thus,

$$Q_{j+1} = Q_j + 4^{-(j+1)} q_{j+1} \geq Q_j = \frac{1}{2}.$$

On the other hand, if $Q_j > \frac{1}{2}$, then since $4^j Q_j \in \mathbb{Z}$, $Q_j \geq \frac{1}{2} + 4^{-j}$ and

$$Q_{j+1} = Q_j + 4^{-(j+1)} q_{j+1} \geq \frac{1}{2} + 4^{-j} + 4^{-(j+1)}(-2) > \frac{1}{2}.$$

The proof of the upper bound is similar.

(b) To prove that $\underline{B}(j+1) \leq R_{j+1} \leq \overline{B}(j+1)$, we shall invoke Lemma 10.14. Let $q = q_{j+1}$. We need only show that

$$q < 2 \Rightarrow m_{q+1}(i, j) \leq U_q(j) \tag{10.26}$$

and

$$q > -2 \Rightarrow L_q(j) + \frac{1}{32} \leq m_q(i, j). \tag{10.27}$$

First suppose $j \leq 2$. If $j = 0$, then $i = 8$ and since $m_q(8, 0) \leq A_0$, (c) implies

$$m_q(8, 0) < 4R_0 + \frac{1}{32} = 4(x - 1) + \frac{1}{32} < \frac{1}{32},$$

which in turn implies $q = q_1 \in \{-2, -1, 0\}$. If $j = 1$, then $i \in \{0, 4, 8\}$. Subject to these constraints, (10.26) and (10.27) may be verified by direct computation for all values of i, j, and q, since $U_q(j)$ and $L_q(j)$ are determined by

$$Q_j = \frac{1}{2} + \frac{i}{16}.$$

Thus, we may assume $j \geq 3$. To prove (10.26) and (10.27), it suffices to show that for all $k \in \{-1, 0, 1, 2\}$,

$$L_k(j) + \frac{1}{32} \leq m_k(i, j) \leq U_{k-1}(j). \tag{10.28}$$

Since

$$\left| \sum_{\ell=3}^{j} 4^{-\ell} q_\ell \right| < 2 \sum_{\ell=3}^{\infty} 4^{-\ell} = \frac{2}{3} 4^{-2} = \frac{1}{24},$$

we have the following bounds on Q_j:

$$\frac{1}{2} + \frac{i}{16} - \frac{1}{24} = \frac{11}{24} + \frac{i}{16} < Q_j < \frac{1}{2} + \frac{i}{16} + \frac{1}{24} = \frac{13}{24} + \frac{i}{16}. \tag{10.29}$$

For the case $i = 1$, a better upper bound is possible. In this case, $Q_2 = \frac{9}{16}$, which implies $q_1 = -2$, $q_2 = 1$, and $Q_1 = \frac{1}{2}$. Since $q_2 < 2$, we must have

$$4R_1 < m_2(0, 1) = \frac{3}{2}$$

and

$$R_1 = 4(x - Q_1^2) = 4\left(x - \frac{1}{4}\right) < \frac{3}{8},$$

which implies $x < \frac{11}{32}$. It follows that $q_3 < 2$, for otherwise

$$4R_2 \geq m_2(1, 2) - \frac{1}{32} = \frac{15}{8} - \frac{1}{32} = \frac{59}{32} > \frac{7}{4},$$

$$R_2 = 4^2(x - Q_2^2) = 16\left(x - \frac{81}{256}\right) > \frac{7}{16},$$

and therefore

$$x > \frac{7}{256} + \frac{81}{256} = \frac{11}{32},$$

a contradiction. Thus, $q_3 \leq 1$ and

$$Q_j < \frac{9}{16} + 4^{-3}q_3 + 2\sum_{\ell=4}^{\infty} 4^{-\ell} \leq \frac{9}{16} + 4^{-3} + \frac{2}{3} \cdot 4^{-3} = \frac{113}{192}.$$

Combining this with (10.29), along with the assumption that $\frac{1}{2} \leq Q_j \leq 1$, we have $Q_{\min}(i) < Q_j < Q_{\max}(i)$, where

$$Q_{\min}(i) = \max\left(\frac{1}{2}, \frac{11}{24} + \frac{i}{16}\right)$$

and

$$Q_{\max}(i) = \begin{cases} \frac{113}{192} & \text{if } i = 1 \\ \min\left(1, \frac{13}{24} + \frac{i}{16}\right) & \text{if } i \neq 1. \end{cases}$$

Applying (10.16) and (10.17) with $r = 4$ and $\rho = \frac{2}{3}$, we conclude that for $k > 0$,

$$U_{k-1}(j) = 2Q_j\left(k - 1 + \frac{2}{3}\right) + 4^{-(j+1)}\left(k - 1 + \frac{2}{3}\right)^2 > 2Q_{\min}(i)\left(k - \frac{1}{3}\right)$$

and

$$L_k(j){=}2Q_j\left(k-\frac{2}{3}\right){+}4^{-(j+1)}\left(k-\frac{2}{3}\right)^2\leq 2Q_{\max}(i)\left(k-\frac{2}{3}\right){+}\frac{1}{256}\left(k-\frac{2}{3}\right)^2,$$

and similarly, for $k \leq 0$,

$$U_{k-1}(j) > 2Q_{\max}(i)\left(k-\frac{1}{3}\right)$$

and

$$L_k(j) \leq 2Q_{\min}(i)\left(k-\frac{2}{3}\right)+\frac{1}{256}\left(k-\frac{2}{3}\right)^2.$$

The required inequality (10.28) follows from these inequalities in all cases. □

Chapter 11
FMA-Based Division

Multiplicative division algorithms are typically based on a sequence of approximations of the reciprocal of the divisor b, derived by the Newton-Raphson method. Given a differentiable function f and a sufficiently accurate initial approximation y_0 of a root of the equation $f(y) = 0$, the Newton-Raphson recurrence formula

$$y_{k+1} = y_k - \frac{f(y_k)}{f'(y_k)}$$

computes a convergent sequence of approximations. For the case

$$f(y) = \frac{1}{y} - b,$$

this yields

$$y_{k+1} = y_k + \frac{\frac{1}{y_k} - b}{\frac{1}{y_k^2}} = y_k(2 - by_k). \tag{11.1}$$

Since the relative error

$$\frac{\frac{1}{b} - y_k}{\frac{1}{b}} = 1 - by_k$$

satisfies

$$1 - by_{k+1} = 1 - by_k(2 - by_k) = (1 - by_k)^2,$$

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_11

the convergence is quadratic, i.e., the number of bits of accuracy of the approximation doubles with each iteration. This means that convergence is achieved in fewer iterations than required by the digit-recurrence approach, but the complexity of each iteration is greater. On the other hand, the hardware requirement may be minimized by utilizing existing multiplication hardware.

The subject of this chapter is a multiplicative technique for floating-point division that was developed for IBM RISC processors in the late 1980s and remains in widespread use today. The initial approximation of the reciprocal of the divisor is derived from tables in read-only memory. The sequence of Newton-Raphson refinements of this approximation is interleaved with a sequence of refinements of the quotient. Central to this process is a hardware *fused multiplication-addition* (FMA) operation, which is assumed to be implemented in support of the standard FMA machine instructions. The significance of this operation is that it has the effect of performing two arithmetic operations with a single rounding, and is therefore both more efficient and more accurate than two separate instructions.

Implementations of this method are generally slower than those based, for example, on the SRT algorithms of Chap. 10, but have the advantage of lower hardware requirements. In fact, the computations are typically performed by either microcode or software.

We shall present proofs of correctness of two representative algorithms based on this approach, which operate on single- and double-precision floating-point numbers, respectively. The inputs to each are a pair of operands, a and b, and a rounding mode, $\mathcal{R}$. The operands are p-exact real numbers in the interval $[1, 2)$, where $p = 24$ or 53. $\mathcal{R}$ may be any of the IEEE rounding modes, although since $\mathcal{R}$ is applied here only to positive arguments, there is no distinction between *RTZ* and *RDN*. The returned value, as specified by IEEE 754 (see Fig. II.1 in Part II), is the rounded quotient $\mathcal{R}(\frac{a}{b}, p)$.

The algorithms are based on two primitive functions, which are assumed to be implemented in hardware:

1. A function *rcp24*, which computes a 24-exact approximation of the reciprocal of a 24-exact number in the interval $[1, 2)$ with relative error bounded by 2^{-23}. The definition of *rcp24*, which is based on table reference and interpolation, is presented in Sect. 11.1.
2. An atomic FMA operation, which computes the rounded value $\mathcal{R}(xy + z, p)$ for p-exact operands x, y, and z and any IEEE rounding mode $\mathcal{R}$. No restriction is imposed on the exponents of the operands.

The relevant theory, which is largely based on the work of Markstein [18] and Harrison [7], is developed in Sects. 11.2 and 11.3. The algorithms are presented in Sect. 11.4 along with proofs of their correctness.

11.1 Reciprocal Approximation

We shall define a function *rcp24* that computes an approximation of the reciprocal of a 24-exact number b, $1 \leq b < 2$, by the method of *minimax quadratic interpolation* [24]. The computation is based on a partition of the interval $[1, 2)$ into 2^k subintervals, $I_i = [1 + 2^{-k}i, 1 + 2^{-k}(i+1))$, where $0 \leq i < 2^k$, and a quadratic function defined on each subinterval. Thus, if $b \in I_i$ and $x = b - (1 + 2^{-k}i)$ is its offset within that subinterval, then the approximation is computed as

$$\frac{1}{b} \approx C_0 + C_1 x + C_2 x^2$$

and rounded to 24 bits, where the coefficients $C_j = C_j(i)$ are read from tables in read-only memory indexed by i, which have been designed to minimize the maximum error over each subinterval. Naturally, the value of k and the precisions of the coefficients are chosen to be as small as possible while providing the desired accuracy of the approximation.

For our present purpose, it has been determined that $k = 7$ and coefficients C_0, C_1, and C_2 of bit-widths 27, 17, and 12, respectively, are sufficient. The resulting tables, which are displayed in Figs. 11.1, 11.2 and 11.3, occupy $2^7(27+17+12)/8 = 896$ bytes of ROM. Since the reciprocal function is positive, decreasing, and convex, $C_0 > 0$, $C_1 < 0$, and $C_2 > 0$. Furthermore, each coefficient satisfies $|C_j| < 1$, i.e., all bits are fractional. Note that the table entries are represented with the radix points and the sign of C_1 omitted. That is, the displayed values are $2^{27}C_0$, $-2^{17}C_1$, and $2^{12}C_2$, all in hexadecimal notation. The approximation is computed as follows:

Definition 11.1 Given a 24-exact number b, $1 \leq b < 2$, let $i = \lfloor 2^7(b-1) \rfloor$ and $x = b - (1 + 2^{-7}i)$. Then

$$rcp24(b) = RNE(C_0(i) + C_1(i)x + C_2(i)x^2, 24),$$

where $C_0(i)$, $C_1(i)$, and $C_2(i)$ are defined as shown in Figs. 11.1, 11.2, and 11.3.

For details pertaining to the construction of such tables and the hardware implementation of the computation of Definition 11.1, the reader is referred to [24]. For our purpose, the following properties of *rcp24* are readily verified by straightforward exhaustive computation, without appealing to the derivation of the tables or the underlying theory:

Lemma 11.1 *If b is 24-exact, $1 \leq b < 2$, and $y_0 = rcp24(b)$, then $\frac{1}{2} < y_0 \leq 1$ and*

$$|1 - by_0| < 2^{-23}.$$

i[3:0] \ i[6:4]	0	1	2	3	4	5	6	7
0	7FFFFFD	71C71C6	6666666	5D1745E	5555556	4EC4EC4	4924924	4444444
1	7F01FC0	70FE3BE	65C393E	5C90A1E	54E4254	4E6470A	48D159D	43FBC04
2	7E07E08	70381C0	6522C3E	5C0B814	54741FA	4E04E06	487EDE0	43B3D5A
3	7D11968	6F74AE0	6483ED0	5B87DDC	5405402	4DA637C	482D1C0	436C82C
4	7C1F079	6EB3E42	63E7062	5B05B06	539782A	4D4873E	47DC120	4325C54
5	7B301EA	6DF5B0C	634C064	5A84F32	532AE22	4CEB917	478BBCE	42DF9BA
6	7A44C6C	6D3A06C	62B2E44	5A05A04	52BF5A8	4C8F8D2	473C1AA	429A044
7	795CEB0	6C80D90	621B97E	5987B1A	5254E7A	4C3463E	46ED290	4254FCE
8	7878786	6BCA1AE	6186186	590B214	51EB852	4BDA130	469EE58	4210842
9	77975B9	6B15C08	60F25DE	588FEA0	51832F2	4B80970	46514DE	41CC984
A	76B981E	6A63BD8	6060606	5816058	511BE18	4B27ED2	4604602	4189376
B	75DED92	69B406A	5FD0180	579D6EC	50B5989	4AD012C	45B81A2	41465FE
C	7507504	6906908	5F417D0	572620A	5050506	4A79049	456C798	4104104
D	7432D64	685B4FC	5EB4882	56B015C	4FEC050	4A22C04	45217C4	40C2470
E	73615A2	67B23A6	5E2931E	563B48C	4F88B2E	49CD430	44D7204	4081020
F	7292CC0	670B450	5D9F736	55C7B4C	4F26566	497889E	448D639	4040404

Fig. 11.1 $2^{27}C_0(i), 0 \leq i < 2^7$

i[3:0] \ i[6:4]	0	1	2	3	4	5	6	7
0	1FFFB	19488	147AC	10ECE	0E38E	0C1E4	0A72E	091A2
1	1F814	18EF8	1439E	10BC0	0E134	0C00A	0A5B3	0906E
2	1F05A	18986	13FA2	108C0	0DEE2	0BE38	0A43C	08F3C
3	1E8CE	18430	13BB8	105CE	0DC9A	0BC6A	0A2CA	08E10
4	1E16C	17EF6	137E2	102E8	0DA5C	0BAA4	0A15E	08CE6
5	1DA36	179D6	1341E	1000C	0D826	0B8E5	09FF6	08BC0
6	1D32A	174D2	1306A	0FD3E	0D5F8	0B72C	09E92	08A9E
7	1CC44	16FE6	12CC8	0FA7C	0D3D4	0B578	09D34	08980
8	1C586	16B12	12936	0F7C4	0D1B6	0B3CC	09BDA	08864
9	1BEEE	16658	125B4	0F51A	0CFA2	0B224	09A84	0874C
A	1B87A	161B4	12242	0F278	0CD94	0B082	09932	08638
B	1B228	15D28	11EE0	0EFE2	0CB8F	0AEE8	097E6	08526
C	1ABFA	158B2	11B8C	0ED56	0C992	0AD50	0969D	08418
D	1A5EE	15450	11848	0EAD6	0C79C	0ABC0	09558	0830E
E	1A002	15006	11510	0E85E	0C5AC	0AA36	09418	08206
F	19A36	14BCE	111E8	0E5F0	0C3C4	0A8B0	092DB	08102

Fig. 11.2 $-2^{17}C_1(i), 0 \leq i < 2^7$

i[3:0] \ i[6:4]	0	1	2	3	4	5	6	7
0	FCF	B1C	81C	618	4B8	3B4	2F4	268
1	F74	AE0	7F8	5FC	4A4	3A4	2EE	264
2	F18	AA8	7D4	5E4	48C	39C	2E0	258
3	EC4	A74	7A8	5CC	47C	388	2D8	254
4	E6A	A40	788	5B8	46C	37C	2D0	24C
5	E18	A08	768	598	45C	372	2C8	244
6	DCC	9DC	744	584	448	368	2BC	23C
7	D7C	9A8	724	570	43C	358	2B4	23C
8	D32	974	704	554	428	350	2AC	230
9	CEB	948	6E4	544	41C	340	2A4	228
A	CA4	918	6C4	52C	408	334	298	224
B	C5C	8F0	6A8	518	3FA	330	294	21C
C	C18	8C4	688	500	3EC	31E	28A	214
D	BD8	894	670	4F0	3E0	314	280	210
E	B98	870	64C	4D8	3CC	310	27C	20C
F	B5C	844	634	4C4	3C0	304	272	208

Fig. 11.3 $2^{12}C_2(i), 0 \le i < 2^7$

11.2 Quotient Refinement

The first step of each algorithm is the computation of an initial approximation y_0 of $\frac{1}{b}$ as an application of the function *rcp24*. An initial approximation of the quotient $\frac{a}{b}$ is then computed as

$$q_0 = RNE(ay_0, p).$$

The accuracy of q_0 may be derived from that of y_0:

Lemma 11.2 *Let $a > 0$, $b > 0$, and $p > 1$. Assume that $|1 - by| \le \epsilon$, and let $q = RNE(ay, p)$. Then*

$$\left|1 - \frac{b}{a}q\right| \le \epsilon + 2^{-p}(1 + \epsilon).$$

Proof Since

$$\left|1 - \frac{b}{a}ay\right| = |1 - by| \le \epsilon$$

and

$$|ay - q| \le 2^{-p}a|y| \le 2^{-p}\frac{a}{b}(1 + \epsilon),$$

$$\left|1 - \frac{b}{a}q\right| \leq \left|1 - \frac{b}{a}ay\right| + \frac{b}{a}|ay - q| \leq \epsilon + 2^{-p}(1 + \epsilon).$$

□

Each of the initial values y_0 and q_0 undergoes a series of refinements, culminating in the final rounded quotient q. Each refinement q_{k+1} of the quotient is computed from the preceding approximation q_k and the current reciprocal approximation y as follows:

$$r_k = RNE(a - bq_k, p)$$

$$q_{k+1} = RNE(q_k + r_k y, p)$$

In the final step, the input rounding mode $\mathcal{R}$ is used instead of *RNE*:

$$r_k = RNE(a - bq_k, p)$$

$$q = \mathcal{R}(q_k + r_k y, p)$$

Our main lemma, due to Markstein [18], ensures that the final quotient is correctly rounded under certain assumptions pertaining to the accuracy of the reciprocal and quotient approximations from which it is derived.

Lemma 11.3 *Let a, b, y, and q be p-exact, where $p > 1$, $1 \leq a < 2$, and $1 \leq b < 2$. Assume that the following inequalities hold:*

(i) $|\frac{a}{b} - q| < 2^{e+1-p}$, *where* $e = \begin{cases} 0 \text{ if } a > b \\ -1 \text{ if } a \leq b; \end{cases}$

(ii) $|1 - by| < 2^{-p}$.

Let $r = a - bq$. Then r is p-exact, and for any IEEE rounding mode $\mathcal{R}$,

$$\mathcal{R}(q + ry, p) = \mathcal{R}\left(\frac{a}{b}, p\right).$$

Proof We may assume $r \neq 0$, for otherwise $\frac{a}{b} = q = q + ry$ and the claim holds trivially. We may also assume $a \neq b$, for otherwise,

$$|1 - q| = \left|\frac{a}{b} - q\right| < 2^{e+1-p} = 2^{-p}$$

implies $q = 1 = \frac{a}{b}$ and $r = 0$. It follows from the bounds on a and b that $e = expo(\frac{a}{b})$. We shall show that $e = expo(q)$ as well.

If $a > b$, then

$$\frac{a}{b} \geq \frac{b + 2^{1-p}}{b} = 1 + \frac{2^{1-p}}{b} > 1 + 2^{-p}$$

and

$$q > \frac{a}{b} - 2^{e+1-p} = \frac{a}{b} - 2^{1-p} > 1 + 2^{-p} - 2^{1-p} = 1 - 2^{-p},$$

which implies $q \geq 1$. On the other hand,

$$q < \frac{a}{b} + 2^{1-p} \leq 2 - 2^{1-p} + 2^{1-p} = 2,$$

and hence, $expo(q) = 0 = e$.

Similarly, if $a < b$, then

$$\frac{a}{b} \geq \frac{1}{2 - 2^{1-p}} = \frac{(1 - 2^{-p}) + 2^{-p}}{2 - 2^{1-p}} = \frac{1}{2} + \frac{2^{-p-1}}{1 - 2^{-p}} > \frac{1}{2} + 2^{-p-1}$$

and

$$q > \frac{a}{b} - 2^{e+1-p} = \frac{a}{b} - 2^{-p} > \frac{1}{2} + 2^{-p-1} - 2^{-p} = \frac{1}{2} - 2^{-p-1},$$

which implies $q \geq \frac{1}{2}$. On the other hand,

$$\frac{a}{b} \leq \frac{a}{a + 2^{1-p}} = 1 - \frac{2^{1-p}}{a + 2^{1-p}} \leq 1 - \frac{2^{1-p}}{2} = 1 - 2^{-p},$$

$$q < \frac{a}{b} + 2^{-p} < 1 - 2^{-p} + 2^{-p} = 1,$$

and $expo(q) = -1 = e$.

To establish p-exactness of r, since

$$|r| = b\left|\frac{a}{b} - q\right| < 2 \cdot 2^{e+1-p} = 2^{e+2-p},$$

either $r = 0$ or $expo(r) \leq e + 1 - p$, and it suffices to show that

$$2^{p-1-(e+1-p)}r = 2^{2p-2-e}r = 2^{2p-2-e}a - 2^{2p-2-e}bq \in \mathbb{Z}.$$

Since a is p-exact, $expo(a) = 0$, and $2p - 2 - e \geq 2p - 2 \geq p - 1$, $2^{2p-2-e}a \in \mathbb{Z}$; since b and q are p-exact, $expo(b) = 0$, and $expo(q) \geq e$,

$$2^{2p-2-e}bq = (2^{p-1}b)(2^{p-1-e}q) \in \mathbb{Z}.$$

For the proof of the second claim, we shall focus on the case $r > 0$; the case $r < 0$ is similar. Let $q' = q + 2^{e+1-p}$. The quotient $\frac{a}{b}$ lies in the interval (q, q'), and its rounded value is either q or q'. For the directed rounding modes (*RUP*, *RDN*, and *RTZ*), we need only show that $q + ry$ also belongs to this interval, i.e., $ry < 2^{1+e-p}$. Since $\frac{a}{b} = q + \frac{r}{b}$, this condition may be expressed as

$$\frac{r}{b} < 2^{e+1-p} \Rightarrow ry < 2^{e+1-p}, \tag{11.2}$$

or

$$2^{p-1-e}r < b \Rightarrow 2^{p-1-e}r < \frac{1}{y}.$$

Since (ii) implies

$$\frac{1}{y} = \frac{b}{by} > \frac{b}{1+2^{-p}},$$

this will follow from

$$2^{p-1-e}r < b \Rightarrow 2^{p-1-e}r \leq \frac{b}{1+2^{-p}}.$$

If $b > 1$, then since $2^{p-1-e}r$ and b are both p-exact and $expo(b) = 0$, we have

$$2^{p-1-e}r < b \Rightarrow 2^{p-1-e}r \leq b - 2^{1-p},$$

and it will suffice to show that

$$b - 2^{1-p} \leq \frac{b}{1+2^{-p}},$$

but this reduces to $b \leq 2 + 2^{1-p}$, and we have assumed that $b < 2$.

On the other hand, if $b = 1$, then

$$2^{p-1-e}r < 1 \Rightarrow 2^{p-1-e}r \leq 1 - 2^{-p},$$

and we need only show that

$$1 - 2^{-p} \leq \frac{1}{1+2^{-p}},$$

which is trivial.

For the remaining case, $\mathcal{R} = RNE$, the proof may be completed by showing that $\frac{a}{b}$ and $q + ry$ lie on the same side of the midpoint $m = q + 2^{e-p}$ of the interval (q, q'). Note that $\frac{a}{b} = m$ is impossible, for if this were true, then since $a = \frac{a}{b} \cdot b$ is p-exact, Lemma 4.14 would imply that $\frac{a}{b} = m$ is also p-exact, but this is not the case. Thus, we must show that

$$\frac{r}{b} < 2^{e-p} \Rightarrow ry < 2^{e-p} \tag{11.3}$$

and

$$\frac{r}{b} > 2^{e-p} \Rightarrow ry > 2^{e-p} \tag{11.4}$$

The proof of (11.3) is the same as that of (11.2), and we may similarly show that (11.4) is a consequence of

$$b + 2^{1-p} \geq \frac{b}{1 - 2^{-p}}.$$

But this is equivalent to $b \leq 2 - 2^{1-p}$, which follows from the assumptions that b is p-exact and $b < 2$. □

IEEE-compliance of the algorithms of interest will be proved as applications of Lemma 11.3 by establishing the two hypotheses (i) and (ii) for appropriate values of y and q.

The next lemma consists of two results. The first specifies the accuracy of an intermediate quotient approximation[1]; the second addresses the final approximation, supplying the first inequality (i) required by Lemma 11.3.

Lemma 11.4 *Let* $1 \leq a < 2$, $1 \leq b < 2$, *and* $p > 0$. *Assume that* $|1 - by| \leq \epsilon$ *and* $|1 - \frac{b}{a} q_0| \leq \delta$. *Let* $r = RNE(a - bq_0, p)$ *and* $q = RNE(q_0 + ry, p)$. *Then*

$$\left|1 - \frac{b}{a} q\right| \leq 2^{-p} + (1 + 2^{-p})\delta\epsilon + 2^{-p}\delta(1 + \epsilon) + 2^{-2p}\delta(1 + \epsilon).$$

If $\delta\epsilon + 2^{-p}\delta(1 + \epsilon) < 2^{-p-1}$, *then*

$$\left|q - \frac{a}{b}\right| < 2^{e+1-p},$$

where $e = \begin{cases} 0 & \text{if } a > b \\ -1 & \text{if } a \leq b. \end{cases}$

Proof Let $u = 1 - by$, $v = 1 - \frac{b}{a} q_0$, $r' = av = a - bq_0$, and $q' = q_0 + ry$. Then

$$q_0 + r'y = \frac{a}{b}(1 - v) + \frac{av}{b}(1 - u) = \frac{a}{b}(1 - uv)$$

and

$$q' = q_0 + r'y + (r - r')y = \frac{a}{b}(1 - uv) + (r - r')y,$$

where

$$|(r - r')y| \leq 2^{-p}|r'y| \leq 2^{-p} \cdot a\delta \cdot \frac{1}{b}(1 + \epsilon) = \frac{a}{b} \cdot 2^{-p}\delta(1 + \epsilon).$$

[1] The first result is not used in the analysis of the algorithms presented in Sect. 11.4, each of which involves only two quotient approximations.

Thus,

$$q' \le \frac{a}{b}(1+\delta\epsilon) + \frac{a}{b} \cdot 2^{-p}\delta(1+\epsilon)$$

and

$$\left|q' - \frac{a}{b}\right| \le \frac{a}{b}(\delta\epsilon + 2^{-p}\delta(1+\epsilon)).$$

For the proof of the first claim, we have

$$\begin{aligned}
\left|q - \frac{a}{b}\right| &\le |q - q'| + \left|q' - \frac{a}{b}\right| \\
&\le 2^{-p}q' + \left|q' - \frac{a}{b}\right| \\
&\le \frac{a}{b}(2^{-p}(1+\delta\epsilon) + 2^{-2p}\delta(1+\epsilon) + \delta\epsilon + 2^{-p}\delta(1+\epsilon)) \\
&= \frac{a}{b}(2^{-p} + (1+2^{-p})\delta\epsilon + \cdot 2^{-p}\delta(1+\epsilon) + \cdot 2^{-2p}\delta(1+\epsilon)).
\end{aligned}$$

For the proof of the second claim, since $\frac{a}{b} \le 2^{e+1}$, we have

$$\left|q' - \frac{a}{b}\right| \le 2^{e+1}(\delta\epsilon + 2^{-p}\delta(1+\epsilon)) < 2^{e+1}2^{-p-1} = 2^{e-p}$$

and

$$\left|q - \frac{a}{b}\right| \le |q - q'| + \left|q' - \frac{a}{b}\right| < 2^{expo(q')-p} + 2^{e-p}.$$

If $expo(q') \le e$, then the claim follows trivially. Thus, we may assume that $expo(q') > e$. But then

$$2^{e+1} \le q' = \frac{a}{b} + \left(q' - \frac{a}{b}\right) < 2^{e+1} + 2^{e-p}$$

implies $q = 2^{e+1}$. It follows that $\frac{a}{b} \le q \le q'$ and

$$\left|q - \frac{a}{b}\right| =\le \left|q' - \frac{a}{b}\right| < 2^{e-p}.$$

□

11.3 Reciprocal Refinement

A refinement y_{k+1} of a given reciprocal approximation y_k may be derived according to (11.1) in two steps:

$$e_k = RNE(1 - by_k, p)$$
$$y_{k+1} = RNE(y_k + e_k y_k, p)$$

Alternatively, as illustrated by the double precision algorithm of Sect. 11.4, an approximation may be computed from the preceding two approximations as follows. This results in lower accuracy but allows the two steps to be executed in parallel:

$$e_{k+1} = RNE(1 - by_{k+1}, p)$$
$$y_{k+2} = RNE(y_k + e_k y_{k+1}, p)$$

The following lemma may be applied to either of these computations.

Lemma 11.5 *Assume that* $|1 - by_1| \leq \epsilon_1$ *and* $|1 - by_2| \leq \epsilon_2$. *Let* $e_1 = RNE(1 - by_1, p)$, *and* $y_3 = RNE(y_3', p)$, *where* $y_3' = y_1 + e_1 y_2$ *and* $p > 0$. *Let*

$$\epsilon_3' = \epsilon_1(\epsilon_2 + 2^{-p}(1 + \epsilon_2)).$$

and

$$\epsilon_3 = \epsilon_3' + 2^{-p}(1 + \epsilon_3').$$

Then (a) $|1 - by_3'| \leq \epsilon_3'$ *and (b)* $|1 - by_3| \leq \epsilon_3$.

Proof Let $u_1 = 1 - by_1$ and $u_2 = 1 - by_2$. Then $|u_1| < \epsilon_1$, $|u_2| < \epsilon_2$, and

$$e_1 = (1 - by_1)(1 + v) = e_1(1 + v),$$

where $|v| \leq 2^{-p}$. Thus,

$$\begin{aligned}
|1 - by_3'| &= |1 - b(e_1 y_2 + y_1)| \\
&= |(1 - by_1) - e_1(by_2)| \\
&= |u_1 - u_1(1 + v)(1 - u_2)| \\
&= |u_1(u_2 + v(u_2 - 1))| \\
&\leq \epsilon_3'
\end{aligned}$$

and

$$|1 - by_3| \leq |1 - by_3'| + b|y_3 - y_3'| \leq \epsilon_3' + 2^{-p}|by_3'| \leq \epsilon_3' + 2^{-p}(1 + \epsilon_3') = \epsilon_3.$$

□

The inequality (b) of Lemma 11.5 provides a significantly reduced error bound for a refined reciprocal approximation y_3 as long as the bounds ϵ_1 and ϵ_2 for the earlier approximations y_1 and y_2 are large in comparison to 2^{-p}. To establish the bound 2^{-p} for the final approximation as required by Lemma 11.3, we shall use the inequality (a), pertaining to the corresponding unrounded value, in conjunction with the following additional lemma, which is a variation by Harrison [7] of another result of Markstein [18]. In practice, the application of this lemma involves explicitly checking a small number of excluded cases.

Lemma 11.6 *Let b be p-exact. Assume that $y = RNE(y', p)$ and $|1 - by'| \leq \epsilon' < 2^{-p-1}$. Let $d = \lceil 2^{2p}\epsilon' \rceil$. Then $|1 - by| < 2^{-p}$, with the possible exceptions $b = 2 - 2^{1-p}k$, $k = 1, \ldots, d$.*

Proof If $b = 1$, then $|1 - y'| < 2^{-p-1}$ implies $y = 1$ and $|1 - by| = 0$. Thus we may assume $b > 1$, and therefore $b \geq 1 + 2^{1-p}$. Consequently,

$$y' \leq \frac{1}{b}(1 + 2^{-p-1}) < \frac{1 + 2^{-p-1}}{1 + 2^{1-p}} < 1,$$

and it follows that $|y - y'| \leq 2^{expo(y')-p} \leq 2^{-p-1}$. Since

$$|1 - by'| \leq \epsilon' = 2^{-2p}2^{2p}\epsilon' \leq 2^{-2p}d,$$

and apart from the allowed exceptions, $b < 2 - 2^{1-p}d$, we have

$$|1 - by| \leq |1 - by'| + b|y - y'| < 2^{-2p}d + (2 - 2^{1-p}d)2^{-p-1} = 2^{-p}.$$

□

11.4 Examples

The single-precision algorithm is given by the following sequence of operations. The spacing of the steps is intended to denote groups of operations that may be executed in parallel. Note that the rounding mode *RNE* is used for all intermediate steps and the input mode $\mathcal{R}$ is used for the final step.

Definition 11.2 $q = divsp(a, b, \mathcal{R})$ is the result of the following sequence of computations:

$$y_0 = rcp24(b)$$

$$q_0 = RNE(ay_0, 24)$$
$$e_0 = RNE(1 - by_0, 24)$$

$$y_1 = RNE(y_0 + e_0 y_0, 24)$$
$$r_0 = RNE(a - bq_0, 24)$$
$$q_1 = RNE(q_0 + r_0 y_1, 24)$$
$$r_1 = RNE(a - bq_1, 24)$$
$$q = \mathcal{R}(q_1 + r_1 y_1, 24)$$

The following lemma, which has been verified by exhaustive computation, will be used in conjunction with Lemma 11.6 in the proof of Theorem 11.1.

Lemma 11.7 *For $k = 1, \ldots, 7$, let $b = 2 - 2^{-23}k$. If y_3 is computed as in Definition 11.2, then $|1 - by_3| < 2^{-24}$.*

Theorem 11.1 *Let a and b be 24-exact with $1 \leq a < 2$ and $1 \leq b < 2$. If $\mathcal{R}$ is an IEEE rounding mode and $q = divsp(a, b, \mathcal{R})$, then*

$$q = \mathcal{R}\left(\frac{a}{b}, 24\right).$$

Proof According to Lemma 11.3, we need only establish the inequalities (ii) $|1 - by_3| < 2^{-24}$ and (i) $|\frac{a}{b} - q_1| < 2^{e-23}$, where e is defined as in the lemma.

Let

$$\begin{aligned}
\epsilon_0 &= 2^{-23}, \\
\epsilon_1' &= \epsilon_0(\epsilon_0 + 2^{-24}(1 + \epsilon_0)), \\
\epsilon_1 &= \epsilon_1' + 2^{-24}(1 + \epsilon_1'), \\
y_1' &= y_0 + e_0 y_0, \\
\text{and} \quad d &= \lceil 2^{48}\epsilon_1' \rceil.
\end{aligned}$$

By Lemma 11.1, $|1 - by_0| \leq \epsilon_0$. By Lemma 11.5 (under the substitutions of y_0 for both y_1 and y_2, ϵ_0 for both ϵ_1 and ϵ_2, and y_1' for y_3'), $|1 - by_1'| \leq \epsilon_1'$ and $|1 - by_1| \leq \epsilon_1$. It is easily verified by direct computation that $\epsilon_1' < 2^{-25}$ and $d = 6$. The required inequality (ii) then follows from Lemmas 11.6 and 11.7.

Let $\delta_0 = \epsilon_0 + 2^{-24}(1 + \epsilon_0)$. By Lemma 11.2,

$$\left|1 - \frac{b}{a} q_0\right| \leq \delta_0.$$

Since $\delta_0\epsilon_1 + 2^{-24}\delta_0(1 + \epsilon_1) < 2^{-25}$ (by direct computation), we may apply Lemma 11.4 (substituting y_1, q_0, ϵ_1, and δ_0 for y, q_2, ϵ, and δ, respectively) to conclude that (i) holds as well. □

The double-precision algorithm follows:

Definition 11.3 $q = divdp(a, b, \mathcal{R})$ is the result of the following sequence of computations:

$$y_0 = rcp24(RTZ(b, 24))$$

$$q_0 = RNE(ay_0, 53)$$
$$e_0 = RNE(1 - by_0, 53)$$

$$r_0 = RNE(a - bq_0, 53)$$
$$y_1 = RNE(y_0 + e_0y_0, 53)$$

$$e_1 = RNE(1 - by_1, 53)$$
$$y_2 = RNE(y_0 + e_0y_1, 53)$$
$$q_1 = RNE(q_0 + r_0y_1, 53)$$

$$y_3 = RNE(y_1 + e_1y_2, 53)$$
$$r_1 = RNE(a - bq_1, 53)$$

$$q = \mathcal{R}(q_1 + r_1y_3, 53)$$

Note that in this case, the initial approximation is based on a truncation of the denominator, which increases its relative error[2]:

Lemma 11.8 *If b is 53-exact, $1 \leq b < 2$, and $y_0 = rcp24(RTZ(b, 24))$, then*

$$|1 - by_0| \leq 2^{-22}.$$

Proof Let $b_0 = RTZ(b, 24)$. Then $b_0 \leq b < b_0 + 2^{-23}$ and by Lemma 11.8,

$$|1 - by_0| \leq |1 - b_0y_0| + |y_0(b - b_0)| \leq 2^{-23} + 2^{-23} = 2^{-22}.$$

□

[2]By exhaustive computation, we could establish an error bound slightly less than $1.5 \cdot 2^{-23}$, which would reduce the number of special cases to be checked from 1027 to 573, but the bound 2^{-22} is sufficient for the proof of Theorem 11.2.

The relative error of the final reciprocal approximation must be computed explicitly in 1027 cases:

Lemma 11.9 *For* $k = 1, \ldots, 1027$*, let* $b = 2 - 2^{-52}k$*. If* y_1 *is computed as in Definition 11.3, then* $|1 - by_1| < 2^{-53}$.

Theorem 11.2 *Let* a *and* b *be 53-exact with* $1 \leq a < 2$ *and* $1 \leq b < 2$*. If* $\mathcal{R}$ *is an IEEE rounding mode and* $q = divdp(a, b, \mathcal{R})$*, then*

$$q = \mathcal{R}\left(\frac{a}{b}, 53\right).$$

Proof Applying Lemma 11.3 once again, we need only establish the two inequalities (ii) $|1 - by_1| < 2^{-53}$ and (i) $|\frac{a}{b} - q_1| < 2^{e-52}$.

Let

$$\begin{aligned}
\epsilon_0 &= 2^{-22}, \\
\epsilon_1' &= \epsilon_0(\epsilon_0 + 2^{-53}(1 + \epsilon_0)), \\
\epsilon_1 &= \epsilon_1' + 2^{-53}(1 + \epsilon_1'), \\
\epsilon_2' &= \epsilon_0(\epsilon_1 + 2^{-53}(1 + \epsilon_1)), \\
\epsilon_2 &= \epsilon_2' + 2^{-53}(1 + \epsilon_2'), \\
\epsilon_3' &= \epsilon_1(\epsilon_2 + 2^{-53}(1 + \epsilon_2)), \\
y_3' &= y_1 + e_1 y_2, \\
\text{and} \quad d &= \lceil 2^{106}\epsilon_3' \rceil.
\end{aligned}$$

Let $b_0 = RTZ(b, 24)$. Then $b_0 \leq b < b_0 + 2^{-23}$ and by Lemma 11.8,

$$|1 - by_0| \leq |1 - b_0 y_0| + |y_0(b - b_0)| \leq 2^{-23} + 2^{-23} = \epsilon_0.$$

By Lemma 11.8, $|1 - by_0| \leq \epsilon_0$. By repeated applications of Lemma 11.5, $|1 - by_1| \leq \epsilon_1$, $|1 - by_2| \leq \epsilon_2$, and $|1 - by_3'| \leq \epsilon_3'$. It is easily verified by direct computation that $\epsilon_3' < 2^{-54}$ and $d = 1027$, and (ii) follows from Lemmas 11.6 and 11.9.

Let $\delta_0 = \epsilon_0 + 2^{-53}(1 + \epsilon_0)$. Direct computation yields $\delta_0\epsilon_1 + 2^{-53}\delta_0(1 + \epsilon_1) < 2^{-54}$, and (i) again follows from Lemma 11.4. □

Part IV
Comparative Architectures: SSE, x87, and Arm

While the principle of correct rounding defines the value of an arithmetic operation under normal conditions, there are a variety of exceptional cases that require special consideration, including invalid and denormal operands, overflow, underflow, and inexact results. Since the advent of floating-point hardware, there has been general agreement on the desirability of an industry standard for exception handling in order to ensure consistent results across all computing platforms. This was the objective of the original IEEE floating-point specification, Standard 754-1985[8], which was developed in parallel with Intel's x87 instruction set, the first "IEEE-compliant" architecture. In the 1990s, a number of competing floating-point architectures emerged. Among these are the *Streaming SIMD (single instruction, multiple data) Extensions*, or *SSE* instructions, which were added by Intel to support multimedia and graphics applications. Of equal importance is the Arm family of reduced instruction set computing architectures, which has dominated the mobile device market.

Unfortunately, these newer architectures did not strictly adhere to IEEE-prescribed behavior. For example, in the event of trapped overflow or underflow, 754-1985 dictates the return of a result generated from the rounded value by a specified shift into the normal range, intended to allow the trap handler to perform scaled arithmetic. This feature of the x87 instructions was not replicated in later architectures, which typically do not return any value in this case.

In 2008, IEEE issued an updated version of Standard 754 [9]. Although it is claimed that "numerical results and exceptions are uniquely determined" by the new standard, an apparent conflicting objective is to accommodate the diverse architectural behaviors that arose in the interim. Consequently, it exhibits a number of ambiguities and deficiencies pertaining, for example, to the detection and handling of underflow, the response to a denormal operand, the order of precedence of the pre-computation conditions, the precedence of operands when more than one is a NaN, and the interaction of exceptions reported by the component operations of a SIMD instruction. Each of these issues has been resolved independently by the

architectures mentioned above with inconsistent results. Consequently, no single standard can possibly serve the needs of a designer or verifier of an implementation of a particular architecture, for which strict backward compatibility is a necessity.

The ACL2 RTL library addresses this problem for the three floating-point architectures of interest—SSE, x87, and Arm—by providing formal executable specifications of the primary elementary arithmetic operations: addition, multiplication, division, square root extraction, and fused multiplication-addition (FMA). These specifications are presented informally but in complete detail in the following chapters. We begin with the SSE instructions (Chap. 12), which have the most commonality with the other two. In our presentation of the x87 and Arm specifications (Chaps. 13 and 14), we emphasize their points of departure from SSE behavior.

Chapter 12
SSE Floating-Point Instructions

The SSE floating-point instructions were introduced by Intel in 1998 and have continually expanded ever since. They operate on single-precision or double-precision data (Definition 5.3) residing in the 128-bit *XMM* registers or the 256-bit *YMM* registers. Some SSE instructions are *packed*, i.e., they partition their operands into several floating-point encodings to be processed in parallel; others are *scalar*, performing a single operation, usually on data residing in the low-order bits of their register arguments. The specifications presented in this chapter apply to both scalar and packed instructions that perform the operations of addition, multiplication, division, square root extraction, and FMA.

A single dedicated 16-bit register, the *MXCSR*, controls and records the response to exceptional conditions that arise during the execution of SSE floating-point operations and controls the rounding of floating-point values.

12.1 SSE Control and Status Register

The MXCSR bits are named as displayed in Fig. 12.1.

- The least significant six bits are the *exception flags*, corresponding to the pre-computation exceptions, invalid operand (IE), denormal operand (DE), and division by zero (ZE); and the post-computation exceptions, overflow (OE), underflow (UE), and inexact result (PE).
- Bit 6 is the *denormal-as-zero* bit, which, if set, coerces all denormal inputs to ± 0.
- Bits 12:7 are the *exception masks* corresponding to the flags, which determine whether an exceptional condition results in the return of a default value or the generation of an exception.

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_12

15	14 13	12	11	10	9	8	7	6	5	4	3	2	1	0
FTZ	RC	PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE

Fig. 12.1 MXCSR: SSE floating-point control and status register

Encoding	Rounding mode
00	*RNE*
01	*RDN*
10	*RUP*
11	*RTZ*

Table 12.1 x86 rounding control

- Bits 14:13 form the *rounding control* field, which encodes a rounding mode as displayed in Table 12.1.
- Bit 15 is the *force-to-zero* bit, which, if set, coerces any denormal output to ± 0.

12.2 Overview of SSE Floating-Point Exceptions

When an exceptional floating-point condition is detected during the execution of an SSE instruction, one of the exception flags MXCSR[5:0] may be set. If a flag is set and the exception is *unmasked*, i.e., the corresponding mask bit in MXCSR[12:7] is 0, then execution of the instruction is terminated, no value is written to the destination, and control is passed to a trap handling routine. If the exception is *masked*, then depending on the exceptional condition, either the instruction proceeds normally or a default value is returned, allowing execution of the program to proceed. For a packed instruction, if any of the component operations results in an unmasked exception, then no result is written for any operation. Otherwise, a result is written for each operation.

Instruction execution consists of three phases: pre-computation, computation, and post-computation. The exceptional conditions are partitioned into two classes, which are detected during the first and third of these phases. The following procedure is followed by both packed and scalar SSE floating-point instructions. Note that in the case of a packed instruction, the procedure is complicated by the requirement of parallel execution:

- Pre-Computation (Sect. 12.3): The operands of each operation are examined in parallel for a set of conditions, some of which result in the setting of a flag, IE, DE, or ZE. If a flag is set and the corresponding mask is clear, then execution is terminated for all operations and no value is written to the destination. Otherwise, for each operation, either a QNaN is selected as a default value (but not yet written), or the computation proceeds.

- Computation (Sect. 12.4): Unless an unmasked exception is detected during the pre-computation phase, for each operation that has not terminated, a computation is performed. If the value is infinite or 0, then a result is determined (but not yet written). Otherwise, execution proceeds.
- Post-Computation (Sect. 12.5): For each remaining operation, the computed value is rounded and the result is examined for a set of conditions, which may result in the setting of one or two of the flags OE, UE, and PE. If a flag is set and the corresponding mask bit is 1, then a result is determined. If a flag is set by any operation and the corresponding mask bit is 0, then an exception is generated and no value is written to the destination for any operation. Otherwise, the result that has been determined for each operation is written.

These three phases and the pre- and post-computation SSE exceptions are discussed in detail in the following sections.

12.3 Pre-computation Exceptions

The first step in the execution of any SSE floating-point instruction, before any exception checking is performed, is to examine the DAZ bit of MXCSR. If this bit is set, then any denormal operand is replaced by a zero of the same sign.

The conditions that may cause an exception flag to be set, or the operation to be terminated with a QNaN value, or both, prior to an SSE computation are as follows:

- SNaN operand: IE is set and the operation is terminated. If the first NaN operand is a QNaN, then the value is that operand; if the first NaN operand is an SNaN, then the value is that operand converted to a QNaN. For this purpose, in the case of an FMA $a \cdot b + c$, the operands are ordered as a, b, c.
- QNaN operand and no SNaN operand: No flag is set, but the operation is terminated. The value is the first NaN operand.
- Undefined Operation: IE is set, the operation is terminated, and the value is the real indefinite QNaN (Definition 5.23). The operands for which this condition holds depends on the operation:
  - Addition: Two infinities with opposite signs;
  - Subtraction: Two infinities with the same sign;
  - Multiplication: Any infinity and any zero;
  - Division: Any two infinities or any two zeroes;
  - Square root extraction: Any operand with negative sign, excluding negative zero;
  - Multiply-accumulate: A product of an infinity and a zero (with no restriction on the other operand), or a product of an infinity and any non-NaN added to an infinity with sign opposite to that of the product.

Exception or termination condition	Flag set	QNaN result (masked case)
SNaN operand	IE	QNaNized operand
QNaN operand	None	Operand
Undefined operation	IE	Indefinite QNaN
Zero exception	ZE	None
Denormal operand	DE	None

Table 12.2 SSE pre-computation exceptions

- A division operation with any zero as divisor and any finite numerical dividend: ZE is set, but the operation proceeds (resulting in an infinity) unless an unmasked exception occurs.
- Any denormal operand (with DAZ = 0) and none of the above conditions: DE is set, but the numerical computation proceeds unless an unmasked exception occurs.

Note that these conditions are prioritized in the order listed: if any condition holds for a given operation, then any other of lower priority is ignored for that operation. For a packed instruction, all operands are examined in parallel for pre-computation exception conditions. Consequently, it is possible for different flags to be set for different operations.

If any exception flag is set during this process and the corresponding mask bit is clear, then all operations are terminated before any computation is performed, no result is written to the destination, and an exception is generated.

If an operation of a packed instruction is terminated with a default value, the value is not written to the destination until execution of the instruction is completed, since no value is written in the event of an unmasked post-computation exception.

The setting of status flags and the default values are summarized in Table 12.2.

12.4 Computation

Unless terminated in response to a pre-computation exceptional condition, each operation of an SSE arithmetic instruction computes an unrounded value, which is then processed according to the contents of the MCXSR and the floating-point format of the instruction as described below. In the case of a packed instruction, all operations for which a default QNaN value has not been determined in the pre-computation stage are similarly processed in parallel.

For each of these operations, a value is computed. If this value is infinite or 0, then no flags are set and the sign of the result is determined by the signs of the operands and the rounding mode $\mathcal{R} = \text{MXCSR}[14:13]$:

- Infinity: The result is an infinity with sign determined according to the operation:
  - Addition: The sign of the infinite operand or operands.
  - Subtraction: The sign of the minuend if it is infinite, and otherwise the inverse of the sign of the subtrahend.
  - Multiplication or division: The product (xor) of the signs of the operands.
  - Square root: The sign of the operand, which must be positive.
  - Multiply-accumulate: The sign of the addend if it is infinite, and otherwise the product (xor) of the signs of the factors.
- Zero: The result is a zero with sign determined according to the operation:
  - Addition: The sign of operands if they agree; if not, then negative if $\mathcal{R} = RDN$, and otherwise positive.
  - Subtraction: The sign of the minuend if it is the inverse of that of the subtrahend; if not, then negative if $\mathcal{R} = RDN$, and otherwise positive.
  - Multiplication or division: The product (xor) of the signs of the operands.
  - Square root: The sign of the operand.
  - Multiply-accumulate: The product of the signs of the factors if it agrees with that of the addend; if not, then negative if $\mathcal{R} = RDN$, and otherwise positive.

Otherwise, execution proceeds to the next phase with the unrounded computed value, which is finite and nonzero.

12.5 Post-Computation Exceptions

The procedure described in this section is applied in the same way to all operations under consideration. For each operation that reaches this phase, the precise mathematical result (which, of course, need not be computed explicitly by an implementation) is a finite nonzero value u, which is rounded according to the rounding mode $\mathcal{R}$ and the precision p of the data format F, producing a value $r = rnd(u, \mathcal{R}, p)$. This value is subjected to the following case analysis, which may result in the setting of one or more exception flags. If any operation produces an unmasked exception, no result is written for any operation. Otherwise, a final result is written for each operation.

- Overflow (r is above the normal range of the target format, i.e., $|r| > lpn(F)$): In all cases, OE is set.
  - Masked Overflow (OM = 1):

    PE is set. The final result, which is valid only if PM = 1, depends on $\mathcal{R}$ and the sign of r.

    If (a) $\mathcal{R} = RNE$, (b) $\mathcal{R} = RUP$ and $r > 0$, or (c) $\mathcal{R} = RDN$ and $r < 0$, then the final result is an infinity with the sign of r.

Otherwise, the result is the encoding of the maximum normal value for the target format, $\pm lpn(F)$, with the sign of r.

- Unmasked Overflow (OM = 0):
  No final result is returned. If $r \neq u$, then PE is set.

• Underflow (r is below the normal range, i.e., $0 < |r| < spn(F)$):

- Masked Underflow (UM = 1):
  If FTZ = 1, then UE and PE are set. The final result, which is valid only if PM = 1, is a zero with the sign of r.
  If FTZ = 0, then u is rounded again to produce $d = drnd(u, \mathcal{R}, F)$, which may be a denormal value, 0, or the smallest normal, $\pm spn(F)$. If $d \neq u$, then both UE and PE are set; otherwise, neither flag is modified. The final result, which is valid unless PE is set and PM = 1, is the encoding of d, with the sign of u if $d = 0$.
- Unmasked Underflow (UM = 0):
  UE is set. No final result is returned. If the $r \neq u$, then PE is set.

• Normal Case (r is within the normal range, i.e., $spn(F) \leq |r| \leq lpn(F)$):
  If $r \neq u$, then PE is set. The final result, which is valid unless PE is set and PM = 1, is the normal encoding of r.

Thus, PE indicates either that the rounded result is an inexact approximation of an intermediate result, or that some other value has been written to the destination. In the case of masked underflow, it may not be obvious that the behavior described above is consistent with the definition of the auxiliary ACL2 function *sse-round*, which sets PE if either $r \neq u$ or $d \neq u$. However, Lemma 6.118 guarantees that if the first inequality holds, then so does the second.

Also note there is one case in which underflow occurs and UE is not modified: UM = 1 and the unrounded value is returned as a denormal.

Chapter 13
x87 Instructions

The x87 instruction set was Intel's first floating-point architecture, introduced in 1981 with the 8087 hardware coprocessor. The architecture provides an array of eight 80-bit data registers, which is managed as a circular stack. Each numerical operand is located either in an x87 data register or in memory and is interpreted according to one of the data formats defined by Definition 5.3. Data register contents are always interpreted according to the double extended precision format (EP). Memory operands may be encoded in the single (SP), double (DP), or double extended precision format. Numerical results are written only to the data registers in the EP format. The architecture also provides distinct 16-bit control and status registers, the FCW and the FSW, corresponding to the single SSE register MXCSR.

The x87 instructions have largely been replaced by the newer SSE architecture, but remain important for applications that require high-precision computations. Our analysis of these instructions benefits from two simplifications relative to the SSE architecture: all instructions are scalar and there is no FMA instruction. Thus, every instruction to be considered here performs a single operation on one or two operands.

13.1 x87 Control Word

The x87 control word, FCW, allows software to manage the precision and rounding of floating-point operations and to control the response to exceptional conditions that may arise during their execution.

The control word bits are named as shown in Fig. 13.1. The least significant six bits are the *exception masks*, which represent the same classes of exceptional conditions that are encoded in the MXCSR: invalid operand (IM), denormal operand (DM), division by zero (ZM), overflow (OM), underflow (UM), and inexact result (PM).

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_13

15	14	13	12	11 10	9 8	7	6	5	4	3	2	1	0
0	0	0	Y	RC	PC	0	1	PM	UM	OM	ZM	DM	IM

Fig. 13.1 x87 control word

Encoding	Precision
00	24
01	
10	53
11	64

Table 13.1 x87 precision control

The 2-bit RC field FCW[11:10] encodes a rounding mode according to the same scheme used for the SSE instructions (Table 12.1).

The control word also includes a 2-bit PC field FCW[9:8], which controls the precision of rounded results. These results are written only to x87 data registers in the EP format, but they are rounded to 24, 32, or 64 bits of precision as determined Table 13.1.

The remaining six bits of FCW are unused. The five bits FCW[15:13] and FCW[7:6] are reserved: any attempt to alter them is ignored. Bit 6 is always set; the other four are always clear. FCW[12], labelled Y and known as the *infinity bit*, may be read or written by software but its value has no pre-defined meaning. This bit was used in interpreting floating-point infinities in pre-386 processors, but is now obsolete.

Note that the x87 control word contains neither a *denormal-as-zero* (DAZ) bit nor a *force-to-zero* (FTZ) bit.

13.2 x87 Status Word

The x87 status word, FSW[15:0], is used by hardware to record exceptional and other conditions that arise during the execution of x87 instructions. It also contains a pointer to the top of the x87 data register stack. The status word bits, as shown in Fig. 13.2, are as follows:

- Exception flags: The least significant seven bits, FSW[6:0], are the exception flags. The six bits FSW[5:0] correspond to the mask bits FCW[5:0] and record exceptional conditions of the six types listed in Sect. 13.1. Bit 6, labelled SF, is set by hardware to indicate a *stack fault*, which may occur during access of the instruction operands (stack underflow) or destination register (stack overflow) prior to execution of the instruction. There is no mask bit in FCW corresponding to SF. Whenever hardware sets SF, it also sets IE.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	TOP			C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

Fig. 13.2 x87 status word

- Exception summary: Bit 7 is the *exception summary* bit ES, indicating an unmasked exception. In fact, this bit is redundant: ES = 1 if and only if at least one of the exception flags FSW[5:0] is set with the corresponding mask bit in FCW[5:0] clear. This invariant is strictly maintained by hardware.
- Condition codes: Bits 8, 9, 10, and 14 are the *condition codes*, C0, C1, C2, and C3, which are modified by x87 instructions. The elementary arithmetic instructions only modify C1, which is cleared by default and set in the event that an instruction returns a result that is larger in absolute value than its precise mathematical value, i.e., the value has been rounded away from 0 or replaced by an infinity.
- Stack top: The three bits FSW[13:11] encode a pointer to the top of the x87 data stack.
- Busy bit: Bit 15 is included only for backward compatibility with the 8087, in which it indicated that the floating-point coprocessor was busy. It is architecturally defined to have the same value as ES.

13.3 Overview of x87 Exceptions

An essential difference between x87 and SSE exceptions is that when an unmasked exceptional condition is detected during execution of an x87 instruction, the exception summary (ES) bit and busy bit (B) are set along with the indicated exception flag, but the exception itself is postponed until the next x87 instruction is encountered. That is, whenever an x87 instruction is initiated, other than the control instructions, the ES bit is examined and if set, execution is aborted and an exception is generated.

In the case of an unmasked pre-computation exception, execution is terminated and, as in the SSE case, no value is written to the destination. For an unmasked post-computation exception, however, in a departure from SSE behavior, execution proceeds and a value is written.

Another distinctive feature of the x87 architecture is the SF flag. The data stack is examined upon initiation of any instruction that requires access to the stack. When a stack fault is detected, SF is set along with IE. If IM = 0, then ES and B are set as well and execution is terminated; otherwise, the real indefinite QNaN is written to the destination. Thus, a stack fault may be viewed as a pre-computation exceptional

condition with priority over all others. However, we shall view the stack access as well as the initial check of ES as extraneous to the execution of an instruction and exclude these features from our formal model.

Thus, we have the following two phases of execution:

- Pre-Computation: The operands are examined for a set of conditions, some of which result in the setting of a flag, IE, DE, or ZE. If a flag is set and the corresponding mask bit, IM, DM, or ZM, is clear, then ES and B are set, execution is terminated, and no value is written to the destination. If no unmasked exception is detected, then either a QNaN is written to the destination or the computation proceeds.
- Computation: Unless execution terminates during the pre-computation phase, a computation is performed. If the value is infinite or 0, then a result is written to the destination. Otherwise, execution proceeds.
- Post-Computation: The computed result is rounded and examined for a set of conditions, which may result in the setting of one or two of the flags OE, UE, and PE. If a flag is set and the corresponding mask bit, OM, UM, or PM, is clear, then ES and B are set. In any case, a value is written to the destination.

13.4 Pre-computation Exceptions

In addition to those already noted, there are three differences between x87 and SSE pre-computation exceptions:

- Unsupported operand: This class of encodings does not exist in the SSE formats. This condition has priority over all other pre-computation exceptions (with the exception of the stack fault). If an unsupported operand (see Definition 5.6) is detected, then IE is set. If IM = 0, then ES and B set as well and execution is terminated; otherwise the real indefinite QNaN is written to the destination.
- NaN operand: For an SNaN or a QNaN input, the priorities and the setting of IE are the same as for SSE instructions, but there are differences in the computation of the default value. First, any SP or DP NaN operand is converted to the EP format by inserting the integer bit and appending the appropriate number of 0s.

  In the case of two NaN operands, the order of the operands is irrelevant. If the operands are distinct NaNs, then the one with the greater significand field is selected, but if the significand fields coincide, then the zero sign field is selected.
- Pseudo-denormal operands (Definition 5.13): This is another class of encodings that does not exist in the SSE formats. A pseudo-denormal operand triggers a denormal exception in the same way as a denormal operand (and in the event of a masked denormal exception, its value is computed by the same formula).

13.5 Post-Computation Exceptions

In the absence of an unmasked exception, the computational and post-computational behavior of the x87 instructions is the same as that of the corresponding SSE instructions with FTZ = 0, except that (a) results are rounded to the precision specified in the FCW and encoded in the double extended precision format, and (b) if a result if rounded away from 0 or replaced by an infinity, then the condition code C1 is set, and otherwise C1 is cleared.

Here we describe the response of an x87 instruction to an unmasked post-computation exceptional condition. We assume that the computation produces a finite nonzero value u, which is rounded according to the rounding mode $\mathcal{R}$ and the precision p indicated by the FCW, producing a value $r = rnd(u, \mathcal{R}, p)$. If r lies outside the normal range, then the objective is to return a normal encoding of a shifted version r' of r from which r can be recovered, and which may be used in computations performed by the trap handler. The shift is intended to produce a result near the center of the normal range in order to minimize the chance of further overflow or underflow.

- Unmasked overflow ($|r| > lpn(EP)$ and OM = 0): OE, ES, and B are set.
  Let $r' = 2^{-3 \cdot 2^{13}} r$. If r' is still above the normal range, then the final result is an infinity with the sign of r; PE and C1 are set.
  Otherwise, the final result is the normal encoding of r'. In this case, PE is set only if $r \neq u$. If $|r| > |u|$, then C1 is set; otherwise, C1 is cleared.
- Unmasked underflow ($0 < |r| < spn(EP)$ and UM = 0): UE, ES, and B are set.
  Let $r' = 2^{3 \cdot 2^{13}} r$. If r' is still below the normal range, then the final result is a zero with the sign of r; PE is set and C1 is cleared.
  Otherwise, the final result is the normal encoding of r'. In this case, PE is set only if $r \neq u$. If $|r| > |u|$, then C1 is set; otherwise, C1 is cleared.
- Unmasked precision exception: The only effect of PM is that if PE is set and PM = 0, then ES and B are set.

Chapter 14
Arm Floating-Point Instructions

The first Arm *Floating-Point Accelerator*, which appeared in 1993, resembled the x87 coprocessor in its use of an 80-bit EP register file. This was succeeded by the *Vector Floating-Point* (VFP) architecture, which included 64-bit registers implementing the single-, double-, and half-precision data formats. The *NEON Advanced SIMD* extension later added 128-bit instructions for media and signal processing applications.

The elementary arithmetic floating-point instructions of the Arm architecture are similar to the corresponding SSE instructions, including both scalar and vector (SIMD) versions. The behavior described in this chapter is common to the VFP and NEON elementary arithmetic instructions. Both architectures provide a floating-point status and control register, *FPSCR*, similar to the SSE MXCSR. A significant difference between SSE and Arm, however, is that the FPSCR does not include exception masks—all instructions are effectively masked. Consequently, an Arm instruction always executes to completion and returns a data result, and there is no interaction between the component operations of a SIMD instruction of the sort described in Sect. 12.2. However, the FPSCR does contain *trap enable* bits corresponding to the flags. If the trap enable bit is set when an exceptional condition occurs, then upon of completion of the instruction, hardware passes control to a trap handler instead of setting the flag.

Since the operations of a SIMD instruction behave independently, we shall confine our attention to the behavior of scalar instructions. The specifications of the instructions presented in this section are formalized by a set of three ACL2 functions, which belong to the library `books/rtl` in the ACL2 repository [13]:

- $\textit{arm-binary-spec}(op, a, b, \textit{FPSCR}, F)$
- $\textit{arm-sqrt-spec}(a, \textit{FPSCR}, F)$
- $\textit{arm-fma-spec}(a, b, c, \textit{FPSCR}, F)$

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_14

Each of these functions takes one or more operands (single-, double-, or half-precision floating-point encodings), the initial contents of the FPSCR register, and the data format F of the instruction, SP, DP, or HP. The function *arm-binary-spec*, which applies to the four binary arithmetic operations, takes an additional argument, *op*, representing the operation (*ADD*, *SUB*, *MUL*, or *DIV*). Each function returns two values: the data result and the final value of the FPSCR.

In Chaps. 16–19, we shall present a set of floating-point modules that have been implemented in an Arm processor. The above functions are the basis of the statements of correctness of these modules.

14.1 Floating-Point Status and Control Register

The FPSCR bits that are relevant to the instructions of interest are named as displayed in Fig. 14.1.

- Bits 4:0 and 7 are the *cumulative exception flags* for invalid operand (IOC), division by zero (DZC), overflow (OFC), underflow (UFC), inexact result (IXC), and denormal operand (IDE),
- Bits 12:8 and 15 are the *trap enables* corresponding to the flags, which determine whether, in the event of exceptional condition, the flag is set by hardware or control is passed to a trap handler. On an implementation that does not support exception trapping, these bits are held at 0.
- Bits 23:22 form the *rounding control* field (RC), which encodes a rounding mode as displayed in Table 14.1. Note the difference between this encoding and that of Table 12.1.
- Bit 24 is the *force-to-zero* bit (FZ), which, if set, coerces both denormal inputs and (except in the half-precision case—see Sect. 14.3) denormal results to ± 0. Thus, this bit plays the roles of both the DAZ and FTZ bits of the SSE MXCSR (Sect. 12.1).
- Bit 25 is the *default NaN* bit (DN). If asserted, any NaN result of an instruction is replaced by the real indefinite QNaN (Definition 5.23).

31	25	24	23 22		15		12	11	10	9	8	7		4	3	2	1	0
	DN	FZ	RC		IDE		IXE	UFE	OFE	DZE	IOE	IDC		IXC	UFC	OFC	DZC	IOC

Fig. 14.1 FPSCR: Arm floating-point status and control register

Encoding	Rounding mode
00	*RNE*
01	*RUP*
10	*RDN*
11	*RTZ*

Table 14.1 Arm rounding control

14.2 Pre-computation Exceptions

The floating-point pre-computation behavior of the Arm architecture is formalized by three functions:

- *arm-binary-pre-comp*$(op, a, b, FPSCR, F)$
- *arm-sqrt-pre-comp*$(a, FPSCR, F)$
- *arm-fma-pre-comp*$(a, b, c, FPSCR, F)$

Each of these returns an optional data value and an updated FPSCR. If a data value is returned, then execution is terminated; otherwise the computation proceeds.

We have noted that in the Arm architecture, a pre-computation exception never prevents the return of a value, and an exception flag is not set unless the corresponding trap enable is clear. The other departures from SSE pre-computation exception handling are in the setting of the denormal flag, the returned value in the case of a NaN operand, and the precedence of an undefined operation over a QNaN operand. Note, however, that the only case in which an undefined operation and a NaN operand can simultaneously occur is an FMA operation with a product of an infinity and a zero with a NaN addend.

The conditions that may cause an exception flag to be set, or the operation to be terminated with a QNaN value, or both, prior to an Arm floating-point computation are as follows:

- Denormal operand: If FZ = 1, then the operand is forced to ± 0 and, unless the format is *HP*, IDC is asserted; otherwise, neither of these actions is taken. The setting of other flags is based on the result of this step. Note that a denormal exception is not suppressed by another exceptional condition; it is possible for two flags to be set.
- SNaN operand: IOC is set. If DN = 1, then the real indefinite QNaN is returned (Definition 5.23). Otherwise, the first SNaN operand is converted to a QNaN and returned. For this purpose, in the case of an FMA $a + b \cdot c$, the operands are ordered as a, b, c.
- Undefined Operation: The conditions are as specified in Sect. 12.3. IOC is set and the real indefinite QNaN is returned.
- QNaN operand and no SNaN operand or undefined operation: If DN = 1, then the real indefinite QNaN is returned. Otherwise, the first NaN operand is returned (with FMA operands ordered as in the SNaN case).
- A division operation with any zero as divisor and any finite numerical dividend: IDZ is set, but the operation proceeds (resulting in an infinity).

14.3 Post-Computation Exceptions

If a final result is not produced during the pre-computation phase, then control is passed to one of the following, which computes an unrounded value:

- $\mathit{arm\text{-}binary\text{-}comp}(op, a, b, \mathit{FPSCR}, F)$
- $\mathit{arm\text{-}sqrt\text{-}comp}(a, \mathit{FPSCR}, F)$
- $\mathit{arm\text{-}fma\text{-}comp}(a, b, c, \mathit{FPSCR}, F)$

If the computed value is infinite or 0, then execution is terminated. No flags are set and the sign of the result is determined by the signs of the operands and the rounding mode $\mathcal{R} = \text{FPSCR}[23:22]$ as described in Sect. 12.4.

Otherwise, the precise mathematical result of the operation is a finite nonzero value u. This value is passed to the common function

$$\mathit{arm\text{-}post\text{-}comp}(u, \mathit{FPSCR}, F),$$

which performs the rounding and detects exceptions as described below. Note that in addition to the absence of exception masks, there are several departures from SSE behavior in the detection and handling of underflow.

Unless u is a denormal, it is rounded according to the rounding mode $\mathcal{R}$ and the precision p of the data format f, producing a value $r = \mathit{rnd}(u, \mathcal{R}, p)$. The returned value and the setting of exception flags are determined by the following case analysis. In all cases, the setting of a flag is understood to be contingent on the value of the corresponding trap enable bit, except in a certain case of underflow as noted below.

- Overflow (r is above the normal range of the target format, i.e., $|r| > \mathit{lpn}(F)$):

  In all cases, OFC and IXC are set. The result depends on $\mathcal{R}$ and the sign of r.

  If (a) $\mathcal{R} = \mathit{RNE}$, (b) $\mathcal{R} = \mathit{RUP}$ and $r > 0$, or (c) $\mathcal{R} = \mathit{RDN}$ and $r < 0$, then the final result is an infinity with the sign of r.

  Otherwise, the result is the encoding of the maximum normal value for the target format, $\pm\mathit{lpn}(F)$, with the sign of r.
- Underflow, which is detected before rounding (u is below the normal range, i.e., $0 < |u| < \mathit{spn}(F)$):

  If FZ = 1 and F is *SP* or *DP*, then UFC is set. (IXC is not set.) UFE is ignored in this case, and FZ is ignored if F is *HP*. The final result is a zero with the sign of u.

  If FZ = 0 or F is *HP*, then u is rounded to produce $d = \mathit{drnd}(u, \mathcal{R}, F)$, which may be a denormal value, 0, or the smallest normal, $\pm\mathit{spn}(F)$. If $d \neq u$, then both UFC and IXC are set; otherwise, neither flag is modified. The final result is the encoding of d, with the sign of u if $d = 0$.
- Normal Case (u and r are both within the normal range):

  If $r \neq u$, then IXC is set. The final result is the normal encoding of r.

Part V
Formal Verification of RTL Designs

The practical significance of the foregoing development is its utility in bringing mathematical analysis and interactive theorem proving to bear on the formal verification of floating-point RTL designs, thereby addressing the inherent limitations of more automatic methods. As a vehicle for combining theorem proving with sequential logic equivalence checking, we have identified a modeling language, essentially a limited subset of C augmented by several C++ class templates. Among these are the register class templates of Algorithmic C [19], an ANSI standard library intended for system and hardware design. The language provides a means of representing a Verilog design in a form that is more compact, abstract, and amenable to formal analysis. It is designed to support a functional programming style, which we have exploited by implementing a translator to the logical language of the ACL2 theorem prover [13]. This provides a path to mechanical verification of the correctness of a model with respect to a high-level formal specification of correctness of the sort presented in Part IV.

Our verification methodology is diagrammed in Fig. V.1. Ideally, the C model is produced in collaboration between a floating-point architect and a verification engineer, in advance of the RTL design. This allows algorithmic bugs to be detected earlier in the design process than is possible with traditional verification methods, which ordinarily cannot begin until stable RTL is available. A special-purpose parser produces the ACL2 translation as well as a more readable pseudocode version, which serves as documentation. The model may be tested through simulation in both C and ACL2, and formally verified by ACL2. The RTL is developed in parallel with the correctness proof. Once the RTL and the proof are both in place, the verification process is completed by sequential logic equivalence checking with a commercial tool, such as Hector [34] or SLEC [20]. This methodology has been used by architects, designers, and verification engineers at Intel and Arm in the specification and formal verification of arithmetic algorithms and their implementations.

In Chap. 15, we summarize the features of the modeling language and its translation to ACL2. The remaining chapters illustrate the methodology with applications to the floating-point operations of multiplication, addition, FMA, division, and

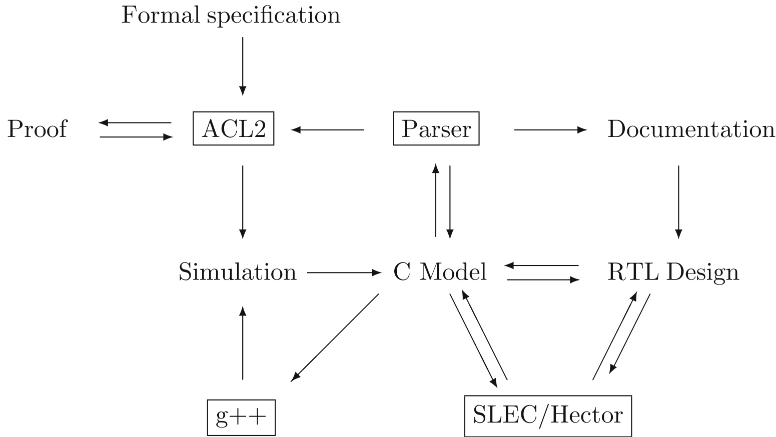


Fig. V.1 Verification methodology

square root extraction, as implemented in the FPU of an Arm Cortex-A class high-end processor. In each case, the corresponding Verilog module has been modeled in our C++ subset, with a code reduction of approximately 85%, and equivalence has been verified with SLEC. The pseudocode versions of the models are included as an appendix. Each model has been translated to ACL2 and mechanically verified to conform to the relevant behavioral specification of Chap. 14. Each of these chapters contains a comprehensive mathematical exposition covering all arithmetic details of the corresponding proof. The remaining relatively trivial aspects of correctness, pertaining to pre-computation exceptions and non-computational special cases, are omitted here but included in the ACL2 proof scripts, which reside in the directory `books/projects/arm/` of the ACL2 repository [13].

Chapter 15
The Modeling Language

The design of our modeling language—its features as well as the restrictions that we impose on it—is driven by the following goals:

- Documentation: C++ is a natural candidate in view of its versatility and widespread use in system modeling. For our purpose as a specification language, we require a subset that is simple enough to allow a clear and easily understood semantic definition, but sufficiently expressive for detailed encoding of complex arithmetic algorithms.
- RTL modeling and equivalence checking: This is the motivation for incorporating the Algorithmic C data types, which model integer and fixed-point registers of arbitrary width and provide the basic bit manipulation features of Verilog, thereby closing the gap between an algorithm and its RTL implementation and easing the burden of equivalence checking. All language features are supported by both Hector [34] and SLEC [20].
- Formal analysis: The objectives of mathematical analysis and translation to ACL2 dictate an applicative programming paradigm, which we promote by eliminating side-effects, replacing the pointers and reference parameters of C++ with other suitable extensions.

The construction of a model in this language is generally a compromise between two opposing objectives. On the one hand, a higher-level model is more susceptible to mathematical analysis and allows a simpler correctness proof. On the other hand, successful equivalence checking of a complex design generally requires a significant amount of proof decomposition, using techniques that depend on structural similarities between the model and the design. As a rule of thumb, the model should be as abstract as possible while performing the same essential computations as the design.

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_15

The language is supported by a special-purpose parser, written in C++ and based on Flex and Bison [16], that performs the following functions:

- Following a check to ensure that a model conforms to the prescribed restrictions, a pseudocode version is generated for the purpose of documentation. This is intended to be more readable than the executable code, especially the arcane syntax of C++ methods. In particular, expressions pertaining to the register classes are replaced with a bit vector notation that is more familiar to Verilog programmers.
- An S-expression representation of the model is generated. This is a first step toward translation to ACL2, which is completed by a translation program written in ACL2 itself.

The parser and ACL2 translator reside in the directory `books/projects/rac/` of the ACL2 repository. Each of the designs presented in the sequel corresponds to a subdirectory of `books/projects/arm/` containing the C++ model, its pseudocode version, the ACL2 translation, and the proof script, which formalizes the proof presented here. The pseudocode models are also collected in an appendix to this book for reference.

The present chapter summarizes the features of the language, assuming a basic understanding of C. For a more thorough treatment of the register classes of Algorithmic C, the reader is referred to [19]. The description of the translator (Sect. 15.6) presupposes familiarity with ACL2 and is not a prerequisite for subsequent chapters.

15.1 Language Overview

Program Structure

A program in our language consists of a sequence of elements of the following three varieties. These may appear in any order, except that an element may not refer to another that follows it in the sequence.

- Type declarations, constructed with the standard C keywords `typedef`, `struct`, and `enum`, each of which associates a type with an identifier.
- Global constant declarations, each of which associates a new constant with an identifier, a type, and a value. Note that global variables are not permitted.
- Function definitions, one of which has special status as the top-level function: its arguments are the inputs of the model, and its return value comprises the outputs. All other functions are called, directly or indirectly, by the top-level function. Recursion (including mutual recursion) is disallowed.

Data Types

All program data are of the following types.

- Three basic numerical types: boolean values (`bool`), unsigned integers (`uint`), and signed integers (`int`).
- The standard C composite types: arrays, structures (`struct`), and enumeration (`enum`) types. Note that since pointers are disallowed, C arrays may not occur as function parameters
- Two class templates of the C++ Standard Template Library [10], both of which are intended to compensate for the absence of pointers and reference parameters: the `array` template, which allows arrays to be passed by value, and the `tuple` template, which provides the effect of multiple-valued functions.
- Integer and fixed-point register types: support for the register class templates of Algorithmic C [19] is provided.

Statements

The body of a function is composed of statements of the following forms:

- Local variable and constant declarations: Note that a program constant may be either global (declared at the top level) or local (declared within the body of a function), while all variables are local.
- Assignments: In another departure from standard C, *assignments*, which are classified as statements, are distinct from *expressions*, which may occur only within statements of the forms listed here.
- The standard C control statements corresponding to the keywords `if`, `if...else`, `for`, `switch`, and `return` (under the limitations specified in Sect. 15.5).
- Statement blocks: arbitrary sequences of statements delineated by "{" and "}".
- Assertions, indicated by the keyword `assert`. An assertion has no semantic import, but may signal a run-time error in either C++ or ACL2.
- Type declarations: As in standard C, data types may be declared globally or locally.

Functions

A function definition consists of the following components:

- A return type, which may be a simple type or an `array` or `tuple` class. Since functions are free of side-effects, every function returns a value; the keyword `void` may not appear in lieu of a return type.

- An identifier, the *name* of the function.
- A list of formal parameters, each of which is represented simply by a type and an identifier, with no qualifiers. In particular, reference parameters are disallowed.
- A statement block, the *body* of the function.

15.2 Parameter Passing

The stipulation that function parameters are passed only by value dictates that native C arrays may be used only as global constants and in instances where an array is used only locally by a function. The effect of passing arrays as value parameters is achieved by means of the standard C++ `array` class template.

As an illustration of its use, suppose that we define a function as follows:

```
array<int, 8> Sum8(array<int, 8> a, array<int, 8> b) {
  for (uint i=0; i<8; i++) {
    a[i] += b[i];
  }
  return a;
}
```

If `a` and `b` are variables of type `array<int, 8>`, then the result of the assignment

```
b = Sum8(a, b);
```

which does not affect the value of `a`, is that each entry of the array `b` is incremented by the corresponding entry of `a`.

Aside from the restriction on parameter passing, there is no semantic difference between ordinary C arrays and instances of an `array` template class. The pseudocode printer, therefore, simply converts `array` objects to C arrays and prints the above definition as follows:

```
int[8] Sum8(int a[8], int b[8]) {
  for (uint i=0; i<8; i++) {
    a[i] += b[i];
  }
  return a;
}
```

The effect of multiple-valued functions is achieved through the `tuple` class template. While the same effect could be achieved by means of an ordinary `struct` return type, this feature provides a convenient means of simultaneously assigning the components of a returned value to local variables of the caller.

For example, the following function performs integer division and returns a quotient and remainder as a tuple:

```
tuple<uint, uint> Divide(m uint, n uint) {
  assert(n != 0);
  uint quot = 0, rem = m;
  while (rem >= n) {
    quot++;
    rem -= n;
  }
  return tuple<uint, uint>(quot, rem);
}
```

A call to this function has the following syntax:

```
uint q, r;
tie(q, r) = Divide(23, 5);
```

The pseudocode printer provides a slightly simpler syntax, printing the above definition as

```
<uint, uint> Divide(m uint, n uint) {
  assert(n != 0);
  uint quot = 0, rem = m;
  while (rem >= n) {
    quot++;
    rem -= n;
  }
  return <quot, rem>;
}
```

and the invocation as

```
uint q, r;
<q, r> = Divide(23, 5);
```

Note that our use of the `tuple` template is intended only for the purpose of parameter-passing and is not recognized by the parser in any other context.

15.3 Registers

The language includes the signed and unsigned integer and fixed-point register class templates of Algorithmic C, which are fully documented on the Mentor Graphics Web site [19]. The unsigned integer registers are the simplest of these and are generally sufficient for modeling any RTL design. The other classes, while not strictly necessary, are sometimes useful in documenting the intended meaning of a register and their more complicated semantics may be convenient in performing arithmetic computations. In the models of Chaps. 16–19, we use signed and unsigned integer registers, but avoid the fixed-point classes.

The register class templates may be instantiated as follows:

- `ac_int<n, false>` and `ac_int<n, true>`: unsigned and signed integer types of width n, where n may be any positive integer;
- `ac_fixed<n, m, false>` and `ac_fixed<n, m, true>`: unsigned and signed fixed-point register types of width n with m integer bits, where n may be any positive integer and m is any integer.

By convention, we use the names `uin`, `sin`, `ufnim`, and `sfnim` for the register types `ac_int<n, false>`, `_int<n, true>`, `ac_fixed<n, m, false>`, and `ac_fixed<n, m, true>`, respectively. These types are expected to be declared as needed at the beginning of a program, as in the following examples:

```
typedef ac_int<96, false> ui96;
typedef ac_int<48, true> si48;
typedef ac_fixed<64, 4, false> uf64i4;
typedef ac_fixed<32, 16, false> uf32i16;
```

A register is associated with two values: a *raw value*, which is a bit vector of the same width as the register, and an integer or rational *interpreted value*. The latter is used when a register is evaluated as an argument of an arithmetic operation or assigned to another variable; the former is used in all other contexts. The interpreted value is derived from the raw value according to the register's type as follows:

- `uin` (unsigned integer): The interpreted value is the same as the raw value, an integer in the interval $[0, 2^n)$.
- `sin` (signed integer): A signed version of `uin`, with the leading bit interpreted as the sign. Thus, the represented range is the interval of integers $[-2^{n-1}, 2^{n-1})$.
- `ufnim` (unsigned fixed-point): Interpreted with an implicit binary point following the most significant m bits. The represented values are rational numbers of the form $2^{m-n}k$, where k is an integer and $0 \leq k < 2^m$.
- `sfnim` (signed fixed-point): A signed version of `ufnim`, representing rational numbers of the form $2^{m-n}k$, where $-2^{m-1} \leq k < 2^{m-1}$.

Thus, the interpreted value of a register of any of these types is related to its raw value r according to Definitions 2.4, 2.5, and 2.7, which are collected here for reference:

- $ui(r) = r$;
- $si(r, n) = \begin{cases} r & \text{if } r < 2^{n-1} \\ r - 2^n & \text{if } r \geq 2^{n-1}. \end{cases}$;
- $uf(r, n, m) = 2^{m-n}ui(r) = 2^{m-n}r$;
- $sf(r, n, m) = 2^{m-n}si(r, n) = \begin{cases} 2^{m-n}r & \text{if } r < 2^{n-1} \\ 2^{m-n}r - 2^m & \text{if } r \geq 2^{n-1}, \end{cases}$.

As discussed in Sect. 2.5, while the width n of a register must be positive, there is no restriction on the number m of integer bits of a fixed-point register. If $m > n$, then the interpreted value is an integer with $m - n$ trailing zeroes; if $m < 0$, then the interpreted value is a fraction with $-m$ leading zeroes.

The Algorithmic C register types include a *bit select* operator and *slice read* and *write* methods. Bit selection is independent of the register type (signed vs. unsigned, integer vs. fixed-point) and uses the familiar syntax. Thus, a bit $x[i]$ of a register x may be extracted or assigned in the natural way. The syntax of bit slices is less straightforward. The slice read method `slc` requires a constant template parameter indicating the width of the slice and a variable argument that represents the base index. Thus,

```
x.slc<4>(n)
```

determines the slice of `x` of width 4 based at index `n`, which is rendered in pseudocode as

```
x[n+3:n].
```

The value of this method is an integer register (independent of whether the register is integer or fixed-point) with the same width parameter as the method and the same sign parameter as the operand. Thus, the numerical value of a slice is a signed or unsigned integer according to the type of the register from which it is extracted.

A bit slice is modified by a method of two arguments, the base index and an integer register containing the value to be written. The width of the slice is not specified explicitly, but rather is inferred from the type of the second argument. For example, the assignment

```
x.set_slc(n, ui4(6))
```

replaces the slice of `x` of width 4 based at index `n` with the value 6 (binary 0110), and is represented in pseudocode as

```
x[n+3:n] = 6.
```

15.4 Arithmetic

The following numerical operators are inherited from native C:

- The unary arithmetic operators `+` and `-` and the binary arithmetic operators `+`, `-`, `*`, and `/`.
- The shift operators, `<<` and `>>`.
- The modulus operator, `%`.
- The bit-wise logical unary complement operator `~` and binary operators `&`, `|`, and `^`.
- The binary boolean-valued arithmetic relational operators `<`, `>`, `<=`, `>=`, `==`, and `!=`.
- The boolean unary operator `!` and binary operators `&&` and `||`.
- The ternary conditional operator `?`.

These operators are extended to integer and fixed-point registers under the restrictions specified in [19]. In addition to these restrictions, we disallow the application

of the division operator (/) to fixed-point arguments. When a register occurs as an argument of an arithmetic or relational operator, its interpreted value is used; when it occurs as an argument of a logical operator, its raw value is used.

Assignment statements may use the basic assignment operator = or any of the assignment operators corresponding to the binary arithmetic and logical operators listed above: `+=`, `*=`, `&=`, etc. An application of either of the unary arithmetic operators `++` and `--` of C (which are not recognized as operators, i.e., may not occur within an expression), is also admitted as an assignment statement. When a numerical value is assigned to a register, it is truncated by discarding fractional and integer bits as required to fit into the register format, as specified in [19].

Arithmetic operations on registers are unbounded and performed with absolute precision. This is consistent with the semantics of ACL2, which is based on unbounded rational arithmetic. However, special care must be taken with arithmetic performed on native C integer data, the inherent limitations of which are not addressed by the ACL2 translator. It is the responsibility of the programmer to avoid arithmetic overflow and match the simpler semantics of ACL2, i.e., to ensure that the results of all such computations and the values of all assignments lie within the bounds of the relevant data formats.

On the other hand, precision may be lost through assignment. When the result of a computation is assigned to an integer or fixed-point register, the least significant fractional bits and most significant integer bits are discarded as necessary for the result to fit into the target format.

15.5 Control Restrictions

The syntax of control statements is constrained in order to minimize the difficulty of translating from an imperative to a functional programming paradigm. Thus, the only control statements supported are those listed in Sect. 15.1. In particular, `while` and `do... while` are not included. Moreover, a number of restrictions are imposed on the supported constructs.

As described in Sect. 15.6, the ACL2 translator converts a `for` loop to an auxiliary recursive function. In order for this function to be admitted into the ACL2 logic, the prover must be able to establish that execution of the function always terminates. To guarantee that this proof succeeds, we require a `for` loop to have the form

```
for (init; test; update) { ... }
```

under the following restrictions:

- *init* is either a variable assignment

  ```
  var = val
  ```

or a declaration

```
type var = val
```

where *var* is the *loop variable*, *val* is a numerical expression, and *type* is either `uint` or `int`.

- *test* is either a comparison between the loop variable and a numerical expression of the form *var op limit*, where *op* is `<`, `<=`, `>`, or `>=`, or a conjunction of the form $test_1$ `&&` $test_2$, where $test_1$ is such a comparison.
- *update* is an assignment to the loop variable *var*. The combination of *test* and *update* must guarantee termination of the loop. The translator derives a `:measure` declaration from *test*, which is used to establish the admissibility of the generated recursive function.

Neither `break` nor `continue` may occur in a `for` loop. In some cases, the loop test may be used to achieve the functionality of `break`. For example, instead of

```
for (uint i=0; i<N; i++) {
  if (expr) break;
  ...
}
```

we may write

```
for (uint i=0; i<N && !expr; i++) {
  ...
}
```

The ACL2 translator converts a `switch` statement to the Lisp `case` macro. We require that each case of a switch have one of two forms:

`case` *label*: $stmt_1 \ldots stmt_k$
`default`: $stmt_1 \ldots stmt_k$

where

(1) if $k > 1$, then `break` does not occur in any $stmt_i$ for $i < k$, and
(2) if $k > 0$, then except for the final case of the statement, $stmt_k$ is `break`.

Further restrictions are imposed on the placement of `return` statements. We require every function body to be a statement block that recursively satisfies the following conditions:

(1) The statement block consists of a non-empty sequence of statements;
(2) None of these statements except the final one contains a `return` statement;
(3) The final statement of the block is either a `return` statement or an `if`...`else` statement of which each branch is a statement block satisfying all three of these conditions.

15.6 Translation to ACL2

Translation of a C++ model to ACL2 is performed in two steps: (1) the parser generates a representation of the model as a set of S-expressions, and (2) an ACL2 program converts this representation to an ACL2 program.

Most of the recognized C primitives correspond naturally to built-in ACL2 functions. The rest are implemented by a set of functions defined in an RTL library book that is included in any book generated by the translator. These include the following:

- `AG` and `AS` extract and set entries of arrays, which are implemented in ACL2 as association lists;
- `BITN`, `BITS`, `SETBITN`, and `SETBITS` access and set bits and slices of bit vectors;
- `LOG=`, `LOG<>`, `LOG<`, `LOG>`, `LOG<=`, and `LOG>=` are boolean comparators corresponding to the C operators `==`, `!=`, `<`, etc., based on the values 1 and 0 instead of the Lisp symbols `T` and `NIL`;
- `LOGIOR1`, `LOGAND1`, and `LOGNOT1` are boolean functions similarly corresponding to the C operators `||`, `&&`, and `!`;
- `IF1` is a macro with the semantics of `IF`, except that it compares its first argument to 0 instead of `NIL`.

The S-expression generated by the parser for a function definition has the form

(`DEFUNC` *name* (arg_1 ... arg_k) *body*)

where *name* is the name of the function, $arg_1, \ldots, arg_k$ are its formal parameters, and *body* is an S-expression derived from its body, which is assumed to be a statement block. The parser generates an S-expression for each statement as follows:

- Statement block: (`BLOCK` $stmt_1 \ldots stmt_k$).
- Simple assignment: (`ASSIGN` *var term*).
- Multiple-value assignment: (`MV-ASSIGN` ($var_1 \ldots var_k$) *term*), where *term* corresponds to a call to a multiple-valued function.
- Variable or constant declaration: (`DECLARE` *var term*) or (`ARRAY` *var term*), where *term* is optional.
- Conditional branch: (`IF` *term left right*), where *left* is a block and *right* is either a block or `NIL`.
- Return statement: (`RETURN` *term*).
- For loop: (`FOR` (*init test update*) *body*), where *init* is a declaration or an assignment, *test* is a term, *update* is an assignment, and *body* is a statement block.
- Switch statement: (`switch` *test* (lab_1 . $stmts_1$) ... (lab_k . $stmts_k$)), where lab_i is either an integer or a list of integers and $stmts_i$ is a list of statements.
- Assertion: (`ASSERT` *fn term*), where *fn* is the name of the function in which the assertion occurs and *term* is a term that is expected to have a nonzero value.

Note that variable types are not explicitly preserved in the translation. Instead, they are used by the parser to inform the translation of terms. Consider, for example, the statement block

```
{ sf8i2 x = -145;
  ui8 y = 100, z = 3;
  z = y[4:2] * x; }
```

In the evaluation of the expression on the right side of the final assignment, the type of `x` dictates that its value is interpreted as a signed rational with 6 fractional bits, and according to the type of `y` and `z`, their assigned values must be truncated to 8 integer bits. Thus, the above code produces the following S-expression:

```
(BLOCK (DECLARE X (BITS (* -145 (EXPT 2 6)) 7 0))
       (LIST (DECLARE Y (BITS 100 7 0))
             (DECLARE Z (BITS 3 7 0)))
       (ASSIGN Z
               (BITS (FL (* (BITS Y 4 2)
                            (/ (SI 8 X) (EXPT 2 6))))
                     7 0)))
```

The translation is completed by an ACL2 program that operates on the output of the parser. The overall strategy of this program is to convert the body of a function to a nest of `LET`, `LET*`, and `MV-LET` terms. For each statement in the body, the translator generates the following:

- *ins*: a list of the variables whose values (prior to execution of the statement) are read by the statement;
- *outs*: a list of the variables (non-local to the statement) that are written by the statement;
- *term*: an expression of which (a) the unbound variables are *ins*, and (b) the value is a multiple value consisting of the updated values of the variables of *outs*, or a single value if *outs* is a singleton.

Each statement except the last corresponds to a level of the nest in which the variables of *outs* are bound to the value of *term*, except that as an optimization to improve readability, adjacent `LET`s are combined into a single `LET` or `LET*` whenever possible. The *term* of the final statement of the body becomes the body of the nest.

As a trivial (and nonsensical) example, the C++ code that generates the pseudocode function

```
uint foo(uint x, uint y, uint z) {
  uint u = y + z, v = u * x;
  <x, y, z> = bar(u, v);
  y = x > y ? 2 * u : v;
  if (x >= 0) {
    u = 2*u;
  }
```

```
    else {
      v = 3 * u;
    }
    if (x < y) {
      return u;
    }
    else {
    return y + v;
    }
  }
```

also generates the corresponding ACL2 function

```
(DEFUN FOO (X Y Z)
  (LET* ((U (+ Y Z)) (V (* U X)))
    (MV-LET (X Y Z) (BAR U V)
      (LET ((Y (IF1 (LOG> X Y) (* 2 U) V)))
        (MV-LET (V U)
          (IF1 (LOG>= X 0)
               (MV V (* 2 U))
               (MV (* 3 U) U))
          (IF1 (LOG< X Y) U (+ Y V)))))))
```

Assertions, which do not affect any program variables, are handled specially. An assertion (`ASSERT` *fn term*) results in a binding of the dummy variable `ASSERT` to the value (`IN-FUNCTION fn term`), where `IN-FUNCTION` is a macro that throws an error if the value of *term* is 0, with a message indicating the function in which the error occurred.

In addition to the top-level ACL2 function corresponding to a C++ function, a separate recursive function is generated for each `for` loop. Its returned values are those of the non-local variables that are assigned within the loop. Its arguments include these variables, along with any variables that are required in the execution of the loop, as well as any variables that occur in the loop initialization or test. The construction of this function is similar to that of the top-level function, but the final statement of the loop body is not treated specially. Instead, the body of the nest of bindings is a recursive call in which the loop variable is replaced by its updated value. The resulting term becomes the left branch of an `IF` expression, of which the right branch is simply the returned variable (if there is only one) or a multiple value consisting of the returned variables (if there are more than one). The test of the `IF` is the test of the loop.

For example, the function

```
uint baz(uint x, uint y, uint z) {
  uint u = y + z, v = u * x;
  for (uint i=0; i<u && u < v; i+=2) {
    v--;
    for (int j=5; j>=-3; j--) {
```

```
        assert(v > 0);
        u = x + 3 * u;
      }
    }
    return u + v;
  }
```

generates three ACL2 functions:

```
(DEFUN BAZ-LOOP-0 (J V X U)
  (DECLARE (XARGS :MEASURE (NFIX (- J (1- -3)))))
  (IF (AND (INTEGERP J) (>= J -3))
      (LET ((ASSERT (IN-FUNCTION BAZ (> V 0)))
            (U (+ X (* 3 U))))
           (BAZ-LOOP-0 (- J 1) V X U))
      U))

(DEFUN BAZ-LOOP-1 (I X V U)
  (DECLARE (XARGS :MEASURE (NFIX (- U I))))
  (IF (AND (INTEGERP I) (INTEGERP U) (INTEGERP V)
           (AND (< I U) (< U V)))
      (LET* ((V (- V 1)) (U (BAZ-LOOP-0 5 V X U)))
            (BAZ-LOOP-1 (+ I 2) X V U))
      (MV V U)))

(DEFUN BAZ (X Y Z)
  (LET* ((U (+ Y Z)) (V (* U X)))
        (MV-LET (V U)
                (BAZ-LOOP-1 0 X V U)
                (+ U V))))
```

Chapter 16
Double-Precision Multiplication

The first illustration of our verification methodology is a proof of correctness of a double-precision floating-point multiplier that supports both the binary FMUL instruction and the ternary FMA. In a typical implementation of the latter, addition is combined with multiplication in a single pipeline by inserting the addend into the multiplier's compression tree as an additional partial product. The resulting FMA latency is somewhat greater than that of a simple multiplication but less than two successive operations. One drawback of this integrated approach is that the computation cannot be initiated until all three operands are available. Another is that in order to conserve area, hardware is typically shared with the operations of pure multiplication and addition, each of which is implemented as a degenerate case of FMA, resulting in increased latencies.

The Arm floating-point design team has pursued the alternative scheme of two distinct operations. Of course, the primary benefit of the FMA instruction is that it performs these operations with a single rounding. Thus, the multiplier produces an unrounded full 106-bit product, which is passed to the adder. The resulting FMA latency is not optimal, but in practice, this operation usually does not occur in isolation [17]. More often, the result of one FMA is passed as an operand to another in the computation of a sum of products $x_1y_1 + \ldots + x_ny_n$, such as a dot product. In this situation, our scheme allows the products to be computed independently and overlapped with the sums, resulting in a lower overall latency.

In this chapter and the next, we state and prove specifications of correctness of a multiplier and an adder pertaining to pure double-precision multiplication and addition, respectively, as well as the support for FMA provided by each. The multiplier is represented by the module `fmul64` of Appendix B, the pseudocode version of a C++ program that was derived from a Verilog RTL design. Functional equivalence between the model and the RTL has been established with SLEC [20]. The model is significantly more compact and readable than the RTL, consisting of 20 kb of code as compared to 140 kb. This reduction is in part a reflection of the more concise syntax of C, but was primarily achieved through the elimination

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_16

of timing and other optimizations. On the other hand, the essential computations performed by the RTL are replicated in the model in order to facilitate the equivalence check. As usual, this involved some experimentation. For example, the multiplier uses a compression tree to reduce a sum of 29 partial products to a pair of vectors, which are then added. In an initial version of the model, this was replaced by a simple sum, but the resulting burden on the tool proved unmanageable. In the final version, the entire tree, consisting of 27 3:2 compressors, is replicated by the function `compress`.

In a similar experiment, the function `CLZ53`, which counts the leading zeroes of a 53-bit vector in logarithmic time by a clever iterative process, was replaced by a simpler and more transparent linear-time version. In this case, the tool was able to manage the complexity of the equivalence check, but the overall execution time suffered markedly, increasing from 2 min to 22 min, and the more faithful version was ultimately used.

Sections 16.1–16.3 discuss the parameters of `fmul64` and the computation of the product and the unrounded sum. Here we limit our analysis to the case of two nonzero numerical operands, as the remaining cases, involving a NaN, an infinity, or a zero operand, are handled trivially by the auxiliary function `specialCase`. In Sects. 16.4 and 16.5, we apply these results to the FMA and FMUL cases, respectively. The results of Sect. 16.4 will be combined with those of Chap. 17 to establish the correctness of the FMA operation.

Notation In our analysis of C models in this and subsequent chapters, we adopt the convention of using italics to denote the mathematical function represented by a function of the model as well as the numerical value of a program variable or expression. If `x` is a variable of a signed integer register type, then x, occurring either in isolation or as an argument of an arithmetic operation, is understood to represent the interpreted value.

16.1 Parameters

The inputs of `fmul64` are as follows:

- `ui64 opa, opb`: Double-precision encodings of the operands.
- `bool fz, dn`: The FZ and DN fields of the FPSCR (Sect. 14.1).
- `ui2 rmode`: The RC field of the FPSCR, a 2-bit encoding of an IEEE rounding mode (Table 14.1), which we shall denote as $\mathcal{R}$.
- `bool fma`: An indication that the operation is FMA rather than FMUL.

Our assumption that *opa* represents a nonzero numerical value means that $opa[62:52] < 2^{11} - 1$, $opa[62:0] \neq 0$, and if $fz = 1$ then $opa[62:52] \neq 0$; the same restrictions apply to *opb*.

The following results are returned:

- `ui117 D`: The data result. In the FMUL case, the double-precision result is $D[63:0]$, with $D[116:64] = 0$; in the FMA case, $D[116]$ is the sign, $D[115:105]$ is the biased exponent, and $D[104:0]$ is the mantissa.
- `ui8 flags`: The exception flags, FPSCR[7 : 0]. We use the mnemonics defined in Fig. 14.1 to refer to the bits of this vector.
- `bool piz`: An indication of a product of an infinity and a zero, producing the default NaN. Valid only for FMA.
- `bool inz`: An indication that the result is an infinity, a NaN, or a zero. Valid only for FMA.
- `bool expOvfl`: An indication that the exponent of the product is too large to be represented in the 11-bit format. Valid only for FMA when $inz = 0$.

Among the local variables computed by *fmul64* are the sign, exponent, and mantissa fields of the operands:

$signa = opa[63]$, $signb = opb[63]$;
$expa = opa[62:52]$, $expb = opb[62:52]$;
$mana = opa[51:0]$, $manb = opb[51:0]$.

We make the following additional definitions:

$$s_A = \begin{cases} 2^{52} + mana \text{ if } expa > 0 \\ mana \text{ if } expa = 0 \end{cases}$$

$$s_B = \begin{cases} 2^{52} + manb \text{ if } expb > 0 \\ manb \text{ if } expb = 0 \end{cases}$$

$$e_A = \begin{cases} expa - (2^{10} - 1) \text{ if } expa > 0 \\ 1 - (2^{10} - 1) \text{ if } expa = 0 \end{cases}$$

$$e_B = \begin{cases} expb - (2^{10} - 1) \text{ if } expb > 0 \\ 1 - (2^{10} - 1) \text{ if } expb = 0 \end{cases}$$

As a simple restatement of the definition of *decode* (Definition 5.16) incorporating the above definitions, the numerical values of the operands are

$$A = (-1)^{signa} 2^{e_A - 52} s_A = decode(opa, DP) \tag{16.1}$$

and

$$B = (-1)^{signb} 2^{e_B - 52} s_B = decode(opb, DP). \tag{16.2}$$

16.2 Booth Multiplier

The 53×53 integer multiplier, as represented by the function *computeProduct*, operates on *mana* and *manb* and takes as additional parameters boolean indications of the conditions $expa = 0$ and $expb = 0$. It returns the product of s_A and s_B.

Lemma 16.1 $prod = s_A \cdot s_B$.

Proof The proof is based on a straightforward application of Corollary 9.4 under the substitutions $n = 53$, $m = 27$, $x = mana$, and $y = manb$, which yields

$$pp[0] + \sum_{i=1}^{26} 2^{2(i-1)} pp[i] \bmod 2^{106} = mana \cdot manb.$$

In order to account for the leading integer bits of s_A and s_B, *computeProduct* inserts two additional terms, $ia[100:0]$ and $ib[101:0]$, defined by

$$ia[100:0] = \begin{cases} 2^{49} manb & \text{if } expa \neq 0 \\ 0 & \text{if } expa = 0, \end{cases}$$

$$ib[100:0] = \begin{cases} 2^{49} mana & \text{if } expb \neq 0 \\ 0 & \text{if } expb = 0, \end{cases}$$

and

$$ib[101] = \begin{cases} 0 \text{ if } expa = expb = 0 \\ 1 \text{ otherwise,} \end{cases}$$

each shifted left by 3 bits. The resulting sum is readily seen to satisfy

$$pp[0] + \sum_{i=1}^{26} 2^{2(i-1)} pp[i] + 2^{52}(ia + ib) \bmod 2^{106} = s_A \cdot s_B.$$

The function *compress* reduces the 29-term sum on the left to a sum of two terms, *ppa* and *ppb*, by means of 27 3:2 compressors. The equivalence of these two sums may be established by 27 simple (although tedious) applications of Lemma 8.4. Thus,

$$\begin{aligned} prod &= ppa + ppb \bmod 2^{106} \\ &= pp[0] + \sum_{i=1}^{26} 2^{2(i-1)} pp[i] + 2^{3}(ia + ib) \bmod 2^{106} \\ &= s_A \cdot s_B. \end{aligned}$$

□

Corollary 16.2 $|AB| = 2^{e_A+e_B-104}prod$.

Proof This follows from (16.1), (16.2), and Lemma 16.1. □

16.3 Unrounded Product

The design represents exponents internally in a 12-bit signed integer format with a bias of -1, which simplifies exponent addition while admitting simple conversion to and from the standard DP exponent format. The C model uses the 12-bit signed integer register type `si12` to encode exponents in this format. The function *expInt* converts an 11-bit DP exponent to internal form. For example, the value encoded by the variable `expaInt` is $expaInt + 1$ (where *expaInt* is its interpreted value). Thus, according to the following lemma, the encoded value is e_A:

Lemma 16.3

(a) $expaInt = e_A - 1$;
(b) $expbInt = e_B - 1$.

Proof If $expa = 0$, then $expInt(expa) = 1 + 2^{11} + 2^{10}$ and

$$\begin{aligned} expaInt &= si(expInt(expa), 12) \\ &= 1 + 2^{11} + 2^{10} - 2^{12} \\ &= 1 - 2^{10} \\ &= e_A - 1. \end{aligned}$$

Similarly, if $expa \neq 0$ and $expa[10] = 0$, then $expInt(expa) = expa + 2^{11} + 2^{10}$ and

$$\begin{aligned} expaInt &= si(expInt(expa), 12) \\ &= expa + 2^{11} + 2^{10} - 2^{12} \\ &= expa - 2^{10} \\ &= e_A - 1. \end{aligned}$$

On the other hand, if $expa[10] = 1$, then $expInt(expa) = expa - 2^{10}$ and again,

$$expaInt = si(expInt(expa), 12) = expa - 2^{10} = e_A - 1.$$

The same proof applies to (b). □

The internal representation of the sum $e_A + e_B$ is *expProdInt*:

Lemma 16.4

(a) $expProdInt = e_A + e_B - 1$*;*
(b) $expBiasedZero = 1 \Leftrightarrow e_A + e_B + (2^{10} - 1) = 0$*;*
(c) $expBiasedNeg = 1 \Leftrightarrow e_A + e_B + (2^{10} - 1) < 0$*.*

Proof By Lemma 16.3,

$$\begin{aligned} expProdInt &\equiv expaInt + expbInt + 1 \\ &\equiv (e_A - 1) + (e_B - 1) + 1 \\ &\equiv e_A + e_B - 1 \pmod{2^{12}}. \end{aligned}$$

Thus, to prove (a), according to Lemma 2.50, we need only show that $-2^{11} \leq e_A + e_B - 1 < 2^{11}$. But this follows from the definitions of e_A and e_B and the assumptions $0 \leq expa \leq 2^{11} - 2$ and $0 \leq expb \leq 2^{11} - 2$.

The claims (b) and (c) follow trivially. □

Depending on the sign of the biased sum $e_A + e_B + (2^{10} - 1)$, one of the functions *rightShft* and *leftShft* is called to perform a shift of the product. If $e_A + e_B + (2^{10} - 1) \leq 0$, then *rightShft* computes the required shift amount as the value of the local variable *expDeficit*:

Lemma 16.5 *If* $e_A + e_B + (2^{10} - 1) \leq 0$*, then*

$$expDeficit = \begin{cases} 2^{10} - 1 & \text{if } expa = expb = 0 \\ 1 - (e_A + e_B + 2^{10} - 1) & \text{otherwise.} \end{cases}$$

Proof The case $expa = expb = 0$ is trivial.

If both $expa > 0$ and $expb > 0$, then

$$\begin{aligned} expDeficit &= (2^{10} - 1 - expa) + (2^{10} - 1 - expb) + 1 + 1 \bmod 2^{10} \\ &= -e_A - e_B + 2 \bmod 2^{10}. \end{aligned}$$

But if $expa = 0$ and $expb > 0$, then

$$\begin{aligned} expDeficit &= (2^{10} - 1) + (2^{10} - 1 - expb) + 1 \bmod 2^{10} \\ &= (2^{10} - 1 - 1) + (2^{10} - expb - 1) + 2 \bmod 2^{10} \\ &= -e_A - e_B + 2 \bmod 2^{10}, \end{aligned}$$

and the same is true if $expb = 0$ and $expa > 0$. Thus, in all cases,

$$expDeficit = -e_A - e_B + 2 \bmod 2^{10} = 1 - (e_A + e_B + 2^{10} - 1) \bmod 2^{10},$$

and we need only show that $0 \leq 1 - (e_A + e_B + 2^{10} - 1) < 2^{10}$. But by hypothesis,

$$1 - (e_A + e_B + 2^{10} - 1) \geq 1,$$

and since $e_A \geq 1 - (2^{10} - 1)$ and $e_B \geq 1 - (2^{10} - 1)$,

$$1 - (e_A + e_B + 2^{10} - 1) \leq 1 - (1 - (2^{10} - 1) + 1 - (2^{10} - 1) + 2^{10} - 1) = 2^{10} - 2.$$

□

When the biased sum is positive and at least one of the operands is denormal, a normalizing left shift may be performed. In this case, the function *CLZ53* counts the leading zeroes of the product:

Lemma 16.6 *If s is a nonzero 53-bit vector, then $CLZ53(s) = 52 - expo(s)$.*

Proof The value of the local variable x is $2^{11}s$. After k iterations of the `for` loop, the value of n is 2^{6-k} and the low n entries of $z[i]$ and $c[i]$ are as follows: Let x_i be the ith slice of x of width 2^k, i.e., $x_i = x[2^k(i+1)-1 : 2^k i]$, and if $x_i \neq 0$, let L_i be the number of leading zeroes of x_i, i.e., $L_i = 2^k - expo(x_i) - 1$. Then for $0 \leq i < n$,

(a) $z[i] = 1 \Leftrightarrow x_i = 0$;
(b) $z[i] = 0 \Rightarrow c[i] = L_i$.

It is easy to see that these properties hold for $k = 0$, and the invariance may be established by a simple inductive argument. Thus, when $k = 6$, $x_0 = x[63 : 0] = x \neq 0$, $z[0] = 0$, and the returned value is

$$c[0] = 63 - expo(x) = 63 - (expo(s) + 11) = 52 - expo(s).$$

□

The left shift amount is determined by *leftShft* according to the value of the local variable *expDiffInt*:

Lemma 16.7 *If $e_A + e_B + (2^{10} - 1) > 0$, then*

$$expDiffInt = e_A + e_B - clz - 1.$$

Proof By Lemma 16.3,

$$expDiffInt \equiv (e_A - 1) + (e_B - 1) - clz + 1 \equiv e_A + e_B - clz - 1 \bmod 2^{12}.$$

Since $0 \leq clz \leq 52$ and

$$2 - 2^{10} \leq e_A + e_B \leq (2^{11} - 2) - (2^{10} - 1) + (2^{11} - 2) - (2^{10} - 1) = 2^{11} - 2,$$

$-2^{11} \leq e_A + e_B - clz - 1 < 2^{11}$ and the claim follows from Lemma 2.50. □

Along with the shifted product, *leftShft* and *rightShft* also return the resulting exponent *expShftInt*. The following observations are easily verified:

Lemma 16.8

(a) $expShftInt \geq -2^{10}$*;*
(b) $expZero = 1 \Leftrightarrow expShftInt = -2^{10}$*;*
(c) $expMax = 1 \Leftrightarrow expShftInt = 2^{10} - 2$*;*
(d) $expInf = 1 \Leftrightarrow expShftInt = 2^{10} - 1$*;*
(e) $expGTinf = 1 \Leftrightarrow expShftInt > 2^{10} - 1$*;*

The additional value *expInc* indicates an overflow condition requiring that *expShftInt* be incremented. We define

$$e_P = expShftInt + expInc + 1, \tag{16.3}$$

so that $expShftInt + expInc$ is the internal representation of e_P. It follows from Lemma 16.8 (a) that $e_P + (2^{10} - 1) \geq 0$. It is a consequence of the following lemma that $e_P + (2^{10} - 1) = 0$ iff the product is subnormal.

Lemma 16.9

(a) If $e_P + (2^{10} - 1) > 0$*, then* $stkFMA = 0$ *and*

$$|AB| = 2^{e_P}(1 + 2^{-105} frac105);$$

(b) If $e_P + (2^{10} - 1) = 0$*, then*

$$2^{-52-(2^{10}-1)} frac105[104:52] \leq |AB| < 2^{-52-(2^{10}-1)}(frac105[104:52] + 1)$$

and

$$|AB| = 2^{-52-(2^{10}-1)} frac105[104:52] \Leftrightarrow frac105[51:0] = stkFMA = 0.$$

Proof The proof is a case analysis based on the sign of the biased exponent sum:
Case 1: $e_A + e_B + (2^{10} - 1) \leq 0$.

In this case, the function *rightShft* is called. We have $expShftInt = -2^{10}$ and $e_P + (2^{10} - 1) = -2^{10} + expInc + 1 + (2^{10} - 1) = expInc$.

If $expDeficit \geq 64$, then $shift \geq 62$, and otherwise $shift = expDeficit$. Thus, by Lemma 16.5, $shift > 0$.

We also have $stkMaskFMA = 2^{shift} - 1$,

$$stkFMA = 0 \Leftrightarrow prod[shift-2:0] = 0 \Leftrightarrow prod0[shift-1:0] = 0,$$

and

$$frac105 = prod0[106:shift][104:0] = \begin{cases} prod0[106:shift] \text{ if } shift > 1 \\ prod0[105:shift] \text{ if } shift = 1. \end{cases}$$

Case 1.1: $expDeficit > 54$.

It follows that $shift > 54$, which implies

$$frac105[104:52] = 0,$$

$$frac105[51:0] = prod0[106:shift],$$

and

$$e_P + (2^{10} - 1) = expInc = 0.$$

Since $AB \neq 0$, $prod \neq 0$, and it follows that $frac105[51:0]$ and $stkFMA$ are not both 0. Thus, we need only show that $|AB| < 2^{-52-(2^{10}-1)}$. By Corollary 16.2, it suffices to show that $e_A + e_B \leq -54 - (2^{10} - 1)$. If $expa = expb = 0$, then

$$e_A + e_B = 1 - (2^{10} - 1) + 1 - (2^{10} - 1) < -54 - (2^{10} - 1).$$

Otherwise, Lemma 16.5 implies

$$e_A + e_B = 1 - (2^{10} - 1) - expDeficit \leq 1 - (2^{10} - 1) - 55 = -54 - (2^{10} - 1).$$

Case 1.2: $expDeficit \leq 54$.
Thus,

$$shift = expDeficit = 1 - (e_A + e_B + 2^{10} - 1) \leq 54.$$

Case 1.2.1: $prod[105] = shift = 1$.
In this case, $e_P + (2^{10} - 1) = expInc = 1$. Since

$$1 - (e_A + e_B + 2^{10} - 1) = shift = 1,$$

$e_A + e_B + 2^{10} - 1 = 0$, which implies $e_P = e_A + e_B + 1$. Furthermore, $prodShft = prod$, $frac105 = prod[104:0]$, and by Corollary 16.2,

$$\begin{aligned}
|AB| &= 2^{e_A+e_B-104} prod \\
&= 2^{e_A+e_B-104}(2^{105} + frac105) \\
&= 2^{e_A+e_B+1}(1 + 2^{-105} frac105) \\
&= 2^{e_P}(1 + 2^{-105} frac105).
\end{aligned}$$

Case 1.2.2: $prod[105] = 0$ or $shift > 1$.
In this case, $e_P + (2^{10} - 1) = expInc = 0$ and we must establish the claims of (b). By Corollary 16.2,

$$|AB| = 2^{e_A+e_B-104} prod$$

$$= 2^{e_A+e_B-105} prod0$$
$$= 2^{e_A+e_B-105} \left(2^{shift+52} prod0[106{:}shift{+}52] + prod0[shift{+}51{:}0]\right)$$
$$= 2^{-52-(2^{10}-1)} \left(prod0[106{:}shift{+}52] + 2^{-(shift+52)} prod0[shift{+}51{:}0]\right).$$

If $shift > 1$, then

$$frac105[104:52] = prod0[106:shift][104:52] = prod0[106:shift+52],$$

and otherwise, $prod0[106] = prod[105] = 0$ and

$$\begin{aligned} frac105[104:52] &= prod0[105:shift][104:52] \\ &= prod0[105:shift+52] \\ &= prod0[106:shift+52]. \end{aligned}$$

Thus,

$$|AB| = 2^{-52-(2^{10}-1)} \left(frac105[104:52] + 2^{-(shift+52)} prod0[shift+51:0]\right),$$

where

$$0 \leq 2^{-(shift+52)} prod0[shift+51:0] < 2^{-(shift+52)} 2^{shift+52} = 1$$

and

$$\begin{aligned} &|AB| = 2^{-52-(2^{10}-1)} frac105[104:52] \\ &\Leftrightarrow prod0[shift+51:0] = 0 \\ &\Leftrightarrow prod0[shift+51:shift] = prod0[shift-1:0] = 0 \\ &\Leftrightarrow frac105[51:0] = stkFMA = 0. \end{aligned}$$

Case 2: $e_A + e_B + (2^{10} - 1) > 0$.

In this case, the function *leftShft* is called. First note that we cannot have $expa = expb = 0$, for this would imply

$$e_A + e_B + (2^{10} - 1) = 1 - (2^{10} - 1) + 1 - (2^{10} - 1) + (2^{10} - 1) = 3 - 2^{10} < 0.$$

Thus,

$$clz = \begin{cases} CLZ53(mana) & \text{if } expa = 0 \\ CLZ53(manb) & \text{if } expb = 0 \\ 0 & \text{otherwise.} \end{cases}$$

It follows from Lemma 16.6 that $expo(s_A) + expo(s_A) = 104 - clz$, and hence $expo(prod)$ is either $104 - clz$ or $105 - clz$.

By Lemma 16.5,

$$expDiffBiasedZero = 1 \Leftrightarrow e_A + e_B - clz - 1 = -2^{10} \Leftrightarrow e_A + e_B + (2^{10} - 1) = clz,$$

$$expDiffBiasedNeg = 1 \Leftrightarrow e_A + e_B - clz - 1 < -2^{10} \Leftrightarrow e_A + e_B + (2^{10} - 1) < clz,$$

and

$$expDiffBiasedPos = 1 \Leftrightarrow e_A + e_B + (2^{10} - 1) > clz.$$

Case 2.1: $e_A + e_B + (2^{10} - 1) > clz$.

In this case,

$$\begin{aligned} e_P &= expShftInt + expInc + 1 \\ &= expDiffInt + expInc + 1 \\ &= e_A + e_B - clz + expInc, \end{aligned}$$

$$e_P + (2^{10} - 1) = e_A + e_B + (2^{10} - 1) - clz + expInc > 0,$$

and we must show that $|AB| = 2^{e_P}(1 + 2^{-105} frac105)$.

Since $shift = clz$,

$$expo(prodShft) = expo(prod) + clz \in \{104, 105\},$$

$ovflMask = 2^{63-clz}$, and

$$expInc = mulOvf = prod[105 - clz] = prodShft[105].$$

Suppose $expInc = 0$. Then $expo(prodShft) = 104$,

$$frac105 = (2prodShft)[104 : 0] = 2prodShft[103 : 0],$$

and

$$\begin{aligned} |AB| &= 2^{e_A + e_B - 104} prod \\ &= 2^{e_A + e_B - clz - 104} prodShft \\ &= 2^{e_A + e_B - clz - 104}(2^{104} + prodShft[103 : 0]) \\ &= 2^{e_A + e_B - clz}(1 + 2^{-105} frac105) \\ &= 2^{e_P}(1 + 2^{-105} frac105). \end{aligned}$$

On the other hand, if $expInc = 1$, then $expo(prodShft) = 105$,

$$frac105 = prodShft[104:0],$$

and

$$\begin{aligned}|AB| &= 2^{e_A+e_B-104}prod \\ &= 2^{e_A+e_B-clz-104}prodShft \\ &= 2^{e_A+e_B-clz-104}(2^{105}+prodShft[104:0]) \\ &= 2^{e_A+e_B-clz+1}(1+2^{-105}frac105) \\ &= 2^{e_P}(1+2^{-105}frac105).\end{aligned}$$

Case 2.2: $e_A + e_B + (2^{10} - 1) = clz$.

In this case, $clz > 0$, $shift = clz - 1$, and

$$mulOvf = prod[105 - shift] = prod[106 - clz] = 0,$$

$$expo(prodShft) = expo(2^{clz-1}prod) \in \{103, 104\},$$

$$expInc = sub2Norm = prod[104 - shift] = prodShft[104],$$

$$|AB| = 2^{e_A+e_B-104}prod = 2^{clz-(2^{10}-1)-104}2^{1-clz}prodShft = 2^{-102-2^{10}}prodShft,$$

$$frac105 = (2prodShft)[104:0] = 2prodShft[103:0],$$

and

$$e_P = expShftInt + expInc + 1 = -2^{10} + expInc + 1.$$

Case 2.2.1: $expInc = 1$.

Since $e_P + (2^{10} - 1) = expInc > 0$, we must show that

$$|AB| = 2^{e_P}(1+2^{-105}frac105).$$

But since $prodShft[104] = expInc = 1$, $expo(prodShft) = 104$ and

$$\begin{aligned}|AB| &= 2^{-102-2^{10}}prodShft \\ &= 2^{-102-2^{10}}(2^{104}+prodShft[103:0]) \\ &= 2^{2-2^{10}}(1+2^{-104}prodShft[103:0]) \\ &= 2^{e_P}(1+2^{-105}frac105).\end{aligned}$$

Case 2.2.2: $expInc = 0$.

Now $e_P + (2^{10} - 1) = expInc = 0$ and we must establish the claims of (b). Since $prodShft[104] = expInc = 0$, $expo(prodShft) = 103$ and

$$frac105 = 2prodShft[103:0] = 2prodShft.$$

We have $shift = clz - 1 = e_A + e_B + 2^{10} - 2$, and hence

$$\begin{aligned}
|AB| &= 2^{e_A+e_B-104}prod \\
&= 2^{shift+2-2^{10}-104}prod \\
&= 2^{-102-2^{10}}(2^{shift}prod) \\
&= 2^{-102-2^{10}}prodShft \\
&= 2^{-103-2^{10}}frac105 \\
&= 2^{-103-2^{10}}(2^{52}frac105[104:52] + frac105[51:0]) \\
&= 2^{-52-(2^{10}-1)}(frac105[104:52] + 2^{-52}frac105[51:0]).
\end{aligned}$$

Since $0 \leq 2^{-52}frac105[51:0] < 1$ and $stkFMA = 0$, the desired result follows.
Case 2.3: $e_A + e_B + (2^{10} - 1) < clz$.

The product is left-shifted by $shift = expProdM1Int \bmod 2^6$, where

$$expProdM1Int = expInt(expa) + expInt(expb) \bmod 2^{12} = e_A + e_B - 2 \bmod 2^{12}.$$

Thus,

$$shift = e_A + e_B - 2 \bmod 2^6 = e_A + e_B + 2^{10} - 2 \bmod 2^6.$$

By assumption, $e_A + e_B + 2^{10} - 1 > 0$ and $e_A + e_B + 2^{10} - 1 < clz \leq 52$. Therefore, $0 \leq e_A + e_B + 2^{10} - 2 < 64$ and, as in Case 2.2.2, $shift = e_A + e_B + 2^{10} - 2$. Furthermore, since $shift < clz - 1$,

$$expo(prodShft) = shift + expo(prod) < (clz - 1) + (105 - clz) = 104,$$

$$expInc = mulOvfl = prod[105 - shift] = 0,$$

$$\begin{aligned}
e_P + (2^{10} - 1) &= expShftInt + expInc + 1 + (2^{10} - 1) \\
&= -2^{10} + 0 + 1 + (2^{10} - 1) \\
&= 0,
\end{aligned}$$

and once again,

$$frac105 = 2prodShft[103:0] = 2prodShft.$$

The proof is completed in Case 2.2.2. □

Lemma 16.10 $e_P + (2^{10} - 1) = exp11 + 2^{11} \cdot expGTinf + expInc.$

Proof We consider the possible values of the leading 2 bits of $expShftInt$: *Case 1*: $expShftInt[11:10] = 0$.

$$exp11 = expShftInt + 2^{10},$$

$$expGTinf = 0,$$

and

$$\begin{aligned} e_P + (2^{10} - 1) &= (expShftInt + expInc + 1) + (2^{10} - 1) \\ &= expShftInt + 2^{10} + expInc \\ &= exp11 + 2^{11} \cdot expGTinf + expInc. \end{aligned}$$

Case 2: $expShftInt[11:10] = 1$.

$$exp11 = expShftInt - 2^{10},$$

$$expGTinf = 1,$$

and

$$\begin{aligned} e_P + (2^{10} - 1) &= (expShftInt + expInc + 1) + (2^{10} - 1) \\ &= expShftInt + 2^{10} + expInc \\ &= exp11 + 2^{11} \cdot expGTinf + expInc. \end{aligned}$$

Case 3: $expShftInt[11:10] = 2$.

This is precluded by Lemma 16.8. *Case 4*: $expShftInt[11:10] = 3$.

$$exp11 = expShftInt - 2^{11} - 2^{10} = expShftInt - 2^{12} + 2^{10},$$

$$expGTinf = 0,$$

and

$$e_P + (2^{10} - 1) = (expShftInt - 2^{12} + expInc + 1) + (2^{10} - 1)$$

$$= expShftInt - 2^{12} + 2^{10} + expInc$$
$$= exp11 + 2^{11} \cdot expGTinf + expInc.$$

□

16.4 FMA Support

Collecting the results of the preceding sections, we have the following main result for the FMA case, which justifies the adder input assumptions postulated in Sect. 17.1.

Lemma 16.11 *Let opa, opb, fz, dn, and rmode be bit vectors of widths 64, 64, 1, 1, and 2, respectively, and let* $fma = 1$. *Assume that if* $fz = 0$, *then each of opa and opb is a normal or a denormal DP encoding, and if* $fz = 1$, *then each is a normal. Let*

$$A = decode(opa, DP),$$

$$B = decode(opa, DP,$$

and

$$\langle D, flags, piz, inz, expOvfl\rangle = fmul64(opa, opb, fz, dn, rmode, fma).$$

The following conditions hold:

(a) $piz = inz = 0$.
(b) *flags is an 8-bit vector with* $flags[k] = 0$ *for all* $k \neq IXC$.
(c) *D is a 117-bit vector with* $D[116] = 1 \Leftrightarrow AB < 0$.
(d) *If* $expOvfl = 1$, *then*

  (i) $|AB| \geq 2^{2^{10}+1}$;
  (ii) $flags[IXC] = 0$.

(e) *If* $expOvfl = 0$ *and* $D[115:105] > 0$, *then*

  (i) $|AB| = 2^{D[115:105]-(2^{10}-1)}(1 + 2^{-105}D[104:0])$;
  (ii) $flags[IXC] = 0$.

(f) *If* $expOvfl = D[115:105] = 0$, *then*

  (i) $2^{-53-(2^{10}-1)}D[104:52] \leq |AB| < 2^{-52-(2^{10}-1)}(D[104:52] + 1)$;
  (ii) $|AB| = 2^{-52-(2^{10}-1)}D[104:52] \Leftrightarrow D[51:0] = flags[IXC] = 0$.

Proof (a), (b), and (c) are trivial.

If $expOvfl = 1$, then either (1) $expGTinf = 1$, which implies $e_P + (2^{10} - 1) \geq 2^{11}$, or (2) $expInf = expInc = 1$, which implies $e_P + (2^{10} - 1) = (2^{11} - 1) + 1 = 2^{11}$. Thus, $e_P \geq 2^{11} - (2^{10} - 1) = 2^{10} + 1$ and (d) follows from Lemma 16.9 (a).

If $expOvfl = 0$, then $expGTinf = 0$ and

$$D[115:105] = exp11 + expInc = e_P + (2^{10} - 1).$$

Since $D[104:0] = frac105$ and $flags[IDC] = stkFMA$, (e) and (f) follow from Lemma 16.9 (a). □

The remaining special cases are characterized by the following result. We omit the proof, which is a straightforward case analysis based on the definition of *fmul64* that has been mechanically checked.

Lemma 16.12 *Let opa, opb, fz, dn, and rmode be bit vectors of widths 64, 64, 1, 1, and 2, respectively, and let* $fma = 1$. *Assume that at least one of opa and opb is a NaN, an infinity, a zero, or a denormal with* $fz = 1$. *Let*

$$\langle D, flags, piz, inz, expOvfl \rangle = fmul64(opa, opb, fz, dn, rmode, fma).$$

The following conditions hold:

(a) $piz = 1 \Leftrightarrow$ *either opa and opb is an infinity and the other is either a zero or a denormal with* $fz = 1$.
(b) $inz = 1$.
(c) $expOvfl = 0$.
(d) flags is an 8-bit vector with

  (i) $flags[IOC] = 1 \Leftrightarrow$ *either opa or opb is an SNaN or* $piz = 1$*;*
  (ii) $flags[IDC] = 1 \Leftrightarrow$ *either opa or opb is a denormal and* $fz = 1$*;*
  (iii) $flags[k] = 0$ *for all* $k \notin \{IOC, IDC\}$.

(e) D is a 117-bit vector with $D[52:0] = 0$ *and*

  (i) If $piz = 1$, *then* $D[116:53]$ *is the real indefinite QNaN;*
  (ii) If either opa or opb is an SNaN, then $D[116:53]$ *is the first SNaN;*
  (iii) If neither opa or opb is an SNaN but at least one is a QNaN, then $D[116:53]$ *is the first QNaN;*
  (iv) If $piz = 0$, *neither opa nor opb is a NaN, and at least one is an infinity, then* $D[116:53]$ *is an infinity with* $D[116] = opa[63]\hat{}opb[63]$*;*
  (v) In the remaining case $D[116:53]$ *is a zero with* $D[116] = opa[63]\hat{}opb[63]$.

16.5 Rounded Product and FMUL

In the FMUL case, rounding begins with the computation of the sticky, guard, and least significant bits of the product:

Lemma 16.13

(a) $stk = 0 \Leftrightarrow frac105[51:0] = stkFMA = 0$;
(b) $grd = frac105[52]$;
(c) $lsb = frac105[53]$.

We refer to the case analysis of the proof of Lemma 16.9.
Case 1: $e_A + e_B + (2^{10} - 1) \leq 0$.
(a) If $shift \leq 55$, then $stkMask = 2^{52+shift} - 1$, $stkMask[106:1] = 2^{51+shift} - 1$, and

$$stk = 0 \Leftrightarrow prod[50 + shift : 0] = 0.$$

But if $shift > 55$, then $stkMask = 2^{107} - 1$, $stk = 1$, $prod[50+shift : 0] = prod \neq 0$, and the same equivalence holds. Thus,

$$\begin{aligned} stk = 0 &\Leftrightarrow prod[50 + shift : 0] = 0 \\ &\Leftrightarrow prod0[51 + shift : 0] = 0 \\ &\Leftrightarrow prod0[51 + shift : shift] = prod0[shift-1 : 0] = 0 \\ &\Leftrightarrow frac105[51 : 0] = stkFMA = 0. \end{aligned}$$

(b) For $k \geq 0$,

$$\begin{aligned} grdMask[k] = 1 &\Leftrightarrow stkMask[106 : 52][k] = 0 \text{ and } stkMask[105 : 51][k] = 1 \\ &\Leftrightarrow stkMask[52 + k] = 0 \text{ and } stkMask[51 + k] = 1 \\ &\Leftrightarrow 52 + k = 52 + shift \\ &\Leftrightarrow k = shift. \end{aligned}$$

Thus, $grdMask = 2^{shift}$ and

$$\begin{aligned} grd = 1 &\Leftrightarrow prod[105 : 51][shift] = 1 \\ &\Leftrightarrow prod[51 + shift] = 1 \\ &\Leftrightarrow prod0[52 + shift] = 1 \\ &\Leftrightarrow frac105[52] = 1. \end{aligned}$$

(c) is similar to (b).

Case 2: $e_A + e_B + (2^{10} - 1) > 0$.

Recall that in this case, $stkFMA = 0$.

(a) Clearly, $stkMask = 2^{52-shift} - 1$. If $mulOvf = 1$, then

$$\begin{aligned} stk = 0 &\Leftrightarrow prod[51 - shift : 0] = 0 \\ &\Leftrightarrow prodShft[51 : 0] = 0 \\ &\Leftrightarrow frac105[51 : 0] = 0, \end{aligned}$$

and if $mulOvf = 0$, then

$$\begin{aligned} stk = 0 &\Leftrightarrow prod[50 - shift : 0] = 0 \\ &\Leftrightarrow prodShft[50 : 0] = 0 \\ &\Leftrightarrow frac105[51 : 0] = 0. \end{aligned}$$

(b) We have $ovfMask = 2^{63-shift}$ and

$$grdMask = ovfMask[63 : 11] = 2^{52-shift}.$$

If $mulOvf = 1$, then

$$\begin{aligned} grd = 1 &\Leftrightarrow prod[52 - shift] = 1 \\ &\Leftrightarrow prodShft[52] = 1 \\ &\Leftrightarrow frac105[52] = 1, \end{aligned}$$

and if $mulOvf = 0$, then

$$\begin{aligned} grd = 1 &\Leftrightarrow prod[51 - shift] = 1 \\ &\Leftrightarrow prodShft[51] = 1 \\ &\Leftrightarrow frac105[52] = 1. \end{aligned}$$

(c) is similar to (b). □

The following allows us to replace the disjunction of *expInc* and *expRndInc*, which appears in the RTL, with their sum:

Lemma 16.14 *If* $expInc = 1$, *then* $expRndInc = 0$.

Proof Suppose $expInc = expRndInc = 1$. Then $fracUnrnd = 2^{52} - 1$. We again refer to the proof of Lemma 16.9.

Case 1: Note that

$$prod \leq (2^{53} - 1)^2 = 2^{106} - 2^{54} + 1 < 2^{106} - 2^{53}.$$

Since $expInc = 1$, $prod[105] = shift = 1$. We have $frac105 = prod[104:0]$ and

$$fracUnrnd = frac105[104:53] = prod[104:53] = 2^{52} - 1.$$

Thus, $prod[105:53] = 2^{53} - 1$ and

$$prod \geq 2^{53}(2^{53} - 1) = 2^{106} - 2^{53},$$

a contradiction.
Case 2: Note that $prod \leq (2^{53} - 1)(2^{53-clz} - 1)$.
Case 2.1: In this case, $shift = clz$,

$$\begin{aligned} prodShft &= 2^{clz} prod \leq 2^{clz}(2^{53} - 1)(2^{53-clz} - 1) \\ &= 2^{106} - 2^{53+clz} - 2^{53} + 2^{clz} < 2^{106} - 2^{53}, \end{aligned}$$

$expInc = prodShft[105] = 1$, $frac105 = prodShft[104:0]$, and

$$fracUnrnd = frac105[104:53] = prodShft[104:53] = 2^{52} - 1.$$

It follows that $prodShft[105:53] = 2^{53} - 1$ and

$$prodShft \geq 2^{53}(2^{53} - 1) = 2^{106} - 2^{53},$$

a contradiction.
Case 2.2: In this case, $clz > 0$, $shift = clz - 1$,

$$\begin{aligned} prodShft &= 2^{clz-1} prod \\ &\leq 2^{clz-1}(2^{53} - 1)(2^{53-clz} - 1) \\ &= 2^{105} - 2^{52+clz} - 2^{52} + 2^{clz-1} \\ &< 2^{105} - 2^{52}, \end{aligned}$$

$expInc = prodShft[104] = 1$, $frac105 = (2prodShft)[104:0]$, and

$$\begin{aligned} fracUnrnd &= frac105[104:53] \\ &= (2prodShft)[104:53] \\ &= prodShft[103:52] \\ &= 2^{52} - 1. \end{aligned}$$

It follows that $prodShft[104:52] = 2^{53} - 1$ and

$$prodShft \geq 2^{52}(2^{53} - 1) = 2^{105} - 2^{52},$$

a contradiction.

Case 2.3: In this case, $expo(prodShft) < 104$, which implies

$$mulOvf = prodShft[105] = 0$$

and

$$sub2Norm = prodShft[104] = 0,$$

contradicting $expInc = 1$. □

We define

$$\mathcal{R}' = \begin{cases} RDN \text{ if } \mathcal{R} = RUP \text{ and } sign = 1 \\ RUP \text{ if } \mathcal{R} = RDN \text{ and } sign = 1 \\ \mathcal{R} \quad \text{otherwise.} \end{cases}$$

Then by Lemma 6.87,

$$\mathcal{R}(AB, 53) = \begin{cases} \mathcal{R}'(|AB|, 53) \text{ if } sign = 0 \\ -\mathcal{R}'(|AB|, 53) \text{ if } sign = 1. \end{cases} \tag{16.4}$$

If AB is subnormal, i.e., $|AB| < spn(DP)$, then denormal rounding is applied instead:

$$drnd(AB, \mathcal{R}, DP) = \begin{cases} drnd(AB, \mathcal{R}', DP) \text{ if } sign = 0 \\ -drnd(AB, \mathcal{R}', DP) \text{ if } sign = 1. \end{cases} \tag{16.5}$$

Lemma 16.15 $|AB| < spn(DP) \Leftrightarrow e_P + (2^{10} - 1) = 0$.

Proof This is an immediate consequence of Lemma 16.9. □

The next two lemmas correspond to the normal and subnormal cases:

Lemma 16.16 *Assume that* $|AB| \geq spn(DP)$ *and let* $r = \mathcal{R}(AB, 53)$.

(a) $r = AB \Leftrightarrow stk = grd = 0$;
(b) $|r| = 2^{e_P - 52 + expRndInc}(2^{52} + fracRnd)$.

Proof We shall invoke Lemma 6.104 with the substitutions $n = 53$,

$$x = 2^{53} + frac105[104 : 52],$$

$$z = 2^{53-e_P}|AB|,$$

and replacing $\mathcal{R}$ with $\mathcal{R}'$. By Lemma 16.9 (a),

$$z = 2^{53}(1 + 2^{-105} frac105)$$

$$= 2^{53} + 2^{-52} \mathit{frac105}$$
$$= 2^{53} + 2^{-52}(2^{52} \mathit{frac105}[104:52] + \mathit{frac105}[51:0])$$
$$= x + 2^{-52} \mathit{frac105}[51:0].$$

Thus, $\lfloor z \rfloor = x$ and by Lemmas 16.9 (a) and 16.13 (a),

$$z \in \mathbb{Z} \Leftrightarrow \mathit{frac105}[51:0] = 0 \Leftrightarrow \mathit{stk} = 0.$$

Since $e = \mathit{expo}(x) = 53$, according to Definition 6.1,

$$\mathit{RTZ}(x, 53) = 2\lfloor 2^{-1}x \rfloor = 2(2^{52} + \mathit{frac105}[104:53]) = 2(2^{52} + \mathit{fracUnrnd})$$

and by Definition 4.3,

$$\mathit{fp}^{+}(\mathit{RTZ}(x, 53), 53) = 2(2^{52} + \mathit{fracUnrnd}) + 2 = 2(2^{52} + \mathit{fracP1}).$$

By Lemma 16.13 (b) and(c),

$$x[e-n:0] = x[e-n] = x[0] = \mathit{frac105}[52] = \mathit{grd}$$

and

$$x[e-n+1] = x[1] = \mathit{frac105}[53] = \mathit{lsb}.$$

By a straightforward case analysis, the conditions under which $\mathcal{R}'(z, 53) = \mathit{fp}^{+}(\mathit{RTZ}(x, 53), 53)$, according to Lemma 6.104, are equivalent to the conditions for $\mathit{rndUp} = 1$.

Thus, (a) follows from Lemma 6.104. For the proof of (b), we consider the following cases.

If $\mathit{rndUp} = 0$, then $\mathit{expRndInc} = 0$ and

$$\mathcal{R}'(z, 53) = \mathit{RTZ}(x, 53)$$
$$= 2(2^{52} + \mathit{fracUnrnd})$$
$$= 2(2^{52} + \mathit{fracRnd}).$$

If $\mathit{rndUp} = 1$, then

$$\mathcal{R}'(z, 53) = \mathit{fp}^{+}(\mathit{RTZ}(x, 53), 53)$$
$$= 2(2^{52} + \mathit{fracP1}).$$

In the latter case, if $\mathit{fracP1} < 2^{52}$, then $\mathit{expRndInc} = 0$ and

$$\mathcal{R}'(z, 53) = 2(2^{52} + \mathit{fracRnd}),$$

and otherwise, $fracP1 = 2^{52}$, $fracRnd = 0$, $expRndInc = 1$, and

$$\mathcal{R}'(z, 53) = 2(2^{52} + 2^{52}) = 4(2^{52}) = 4(2^{52} + fracRnd).$$

Thus, in all cases,

$$\mathcal{R}'(2^{53-e_P}|AB|, 53) = \mathcal{R}'(z, 53) = 2^{1+expRndInc}(2^{52} + fracRnd)$$

and by Eq. (16.4),

$$|r| = \mathcal{R}'(|AB|, 53) = 2^{e_P-52+expRndInc}(2^{52} + fracRnd).$$

□

Lemma 16.17 *Assume $|AB| < spn(DP)$ and let $d = drnd(AB, \mathcal{R}, DP)$.*

(a) $d = AB \Leftrightarrow stk = grd = 0$;

(b) $|d| = \begin{cases} 2^{1-(2^{10}-1)}(1 + 2^{-52}fracRnd) & \text{if } expRndInc = 1 \\ 2^{-51-(2^{10}-1)}fracRnd & \text{if } expRndInc = 0. \end{cases}$

Proof We shall invoke Lemma 6.104 with the substitutions $n = 53$,

$$x = 2^{53} + frac105[104 : 52],$$

$$z = 2^{52+(2^{10}-1)}(|AB| + 2^{1-(2^{10}-1)}),$$

and replacing $\mathcal{R}$ with $\mathcal{R}'$. By Lemma 16.9 (b),

$$frac105[104 : 52] \leq 2^{52+(2^{10}-1)}|AB| < frac105[104 : 52] + 1,$$

and hence,

$$x = 2^{53} + \left\lfloor 2^{52+(2^{10}-1)}|AB| \right\rfloor = \left\lfloor 2^{53} + 2^{52+(2^{10}-1)}|AB| \right\rfloor = \lfloor z \rfloor.$$

By Lemmas 16.9 (b) and 16.13 (a),

$$z \in \mathbb{Z} \Leftrightarrow frac105[51 : 0] = stkFMA = 0 \Leftrightarrow stk = 0.$$

As in the proof of Lemma 16.16,

$$\mathcal{R}'(z, 53) = z \Leftrightarrow stk = grd = 0$$

and

$$\mathcal{R}'(z, 53) = 2^{1+expRndInc}(2^{52} + fracRnd),$$

which implies

$$\begin{aligned}\mathcal{R}'(|AB| + 2^{1-(2^{10}-1)}, 53) &= 2^{-52-(2^{10}-1)}\mathcal{R}'(z, 53)\\ &= 2^{-52-(2^{10}-1)}2^{1+expRndInc}(2^{52} + fracRnd)\\ &= 2^{-51-(2^{10}-1)+expRndInc}(2^{52} + fracRnd).\end{aligned}$$

By Eq. (16.5) and Lemma 6.107,

$$|d| = drnd(|AB|, \mathcal{R}', DP) = \mathcal{R}'(|AB| + 2^{1-(2^{10}-1)}), 53) - 2^{1-(2^{10}-1)},$$

and (a) follows easily. To complete the proof of (b), we note that if $expRndInc = 1$, then $fracRnd = 0$ and

$$|d| = 2^{2-(2^{10}-1)} - 2^{1-(2^{10}-1)} = 2^{1-(2^{10}-1)} = 2^{1-(2^{10}-1)}(1 + 2^{-52}fracRnd),$$

and if $expRndInc = 0$, then

$$|d| = 2^{-51-(2^{10}-1)}(2^{52} + fracRnd) - 2^{1-(2^{10}-1)} = 2^{-51-(2^{10}-1)}fracRnd.$$

□

Lemma 16.18

(a) $|\mathcal{R}(AB, 53)| < spn(DP) \Leftrightarrow underflow = 1$.
(b) $|\mathcal{R}(AB, 53)| > lpn(DP) \Leftrightarrow overflow = 1$.

Proof

(a) By Lemma 16.8 (b) and Eq. (16.3),

$$\begin{aligned}e_P + (2^{10} - 1) = 0 &\Leftrightarrow expShftInt = -2^{10} \text{ and } expInc = 0\\ &\Leftrightarrow expZero = 1 \text{ and } expInc = 0\\ &\Leftrightarrow underflow = 1.\end{aligned}$$

(b) We may assume $|AB| > spn(DP)$, and hence $e_P + (2^{10} - 1) > 0$. By Lemma 16.16, $expo(\mathcal{R}(AB, 53)) = e_P + expRndInc$. The claim follows from Eq. (16.3) and Lemmas 16.8 and 16.14.

□

Lemma 16.19 *Assume that* $|AB| \geq spn(DP)$ *and let* $r = \mathcal{R}(AB, 53) \leq lpn(DP)$. *Then*

$$D = nencode(r, DP).$$

Proof By Lemmas 16.16 and 16.10,

$$\begin{aligned}
expo(r) &= e_P + expRndInc \\
&= exp11 + 2^{11} \cdot expGTinf + expInc - (2^{10} - 1) + expRndInc \\
&\leq expo(lpn(DP)) \\
&= 2^{10} - 1,
\end{aligned}$$

which implies $expGTinf = 0$ and

$$\begin{aligned}
|r| &= 2^{e_P + expRndInc}(1 + 2^{-52} fracRnd) \\
&= 2^{exp11 + + expInc - (2^{10} - 1) + expRndInc}(1 + 2^{-52} fracRnd) \\
&= 2^{expRnd - (2^{10} - 1)}(1 + 2^{-52} fracRnd).
\end{aligned}$$

It is clear that

$$sign = \begin{cases} 0 \text{ if } r > 0 \\ 1 \text{ if } r < 0, \end{cases}$$

and the claim follows from Definition 5.8 and Lemma 5.2. □

Lemma 16.20 *Assume* $|AB| < spn(DP)$ *and* $fz = 0$. *Let* $d = drnd(AB, \mathcal{R}, DP)$.

(a) If $d = 0$*, then* $D = zencode(sign, DP)$*;*
(b) If $|d| = spn(DP)$*, then* $D = nencode(d, DP)$*;*
(c) If $0 < |d| < spn(DP)$*, then* $D = dencode(d, DP)$.

Proof By Lemma 16.10, $exp11 = expInc = 0$, and hence $expRnd = expRndInc$. If $d = 0$, then we must have $expRnd = fracRnd = 0$, and (a) follows trivially, while the cases $expRnd = 1$ and $expRnd = 0$ correspond (b) and (c), respectively. □

Our correctness theorem for FMUL matches the behavior of the top-level function *fmul64* with the specification function *arm-binary-spec* of Chap. 14. The arguments of the latter are a binary operation (*ADD*, *SUB*, *MUL*, or *DIV*), two operands, the initial FPSCR, and a FP format, and its returned values are the data result and the updated FPSCR. To account for the different interface of *fmul64*, the *FZ*, *DN*, and rounding control bits are extracted from the FPSCR, which is ultimately combined with the *flags* output.

For the trivial cases involving a zero, infinity, or NaN operand, the proof is a straightforward comparison of the function *specialCase* of the model with the specification functions *arm-binary-pre-comp* and *arm-binary-comp*. For the remaining computational case, the theorem similarly follows from a simple comparison of *fmul64* and *arm-post-comp* and the application of Lemmas 16.16–16.20.

Theorem 16.1 *Let opa and opb be 64-bit vectors, let R_{in} be a 32-bit vector with $fz = R_{in}[24]$, $dn = R_{in}[25]$, and $rmode = R_{in}[23:22]$, and let $fma = 0$. Let*

$$\langle D_{spec}, R_{spec} \rangle = \textit{arm-binary-spec}(MUL, opa, opb, R_{in}, DP)$$

and

$$\langle D, flags, piz, inz, expOvfl \rangle = fmul64(opa, opb, fz, dn, rmode, fma).$$

Then $D[63:0] = D_{spec}$ and $R_{in} \mid flags = R_{spec}$.

Chapter 17
Double-Precision Addition and FMA

The double-precision addition module `fadd64` supports both the binary FADD instruction and, in concert with the multiplier, the ternary FMA instruction. Thus, both operands may be simple DP encodings, or one of them may be an unrounded product in the format of the data output of `fmul64` as described in Chap. 16.

As is evident in the pseudocode version of the adder displayed in Appendix C, the technique of leading zero anticipation (LZA) of Sect. 8.2 is central to its design. While the timing details of the RTL are largely obscured in the model, the computation is performed in two cycles. In the first cycle, the LZA logic is executed concurrently with the right shift and the addition, which is followed by the normalizing left shift. (In this design, the left shift is not performed in advance of the addition as suggested by Fig. 8.7.) Exponent computation, rounding, and the detection of post-computation exceptions are done in the second cycle.

The trivial cases of NaN and infinite inputs and a zero sum are handled by separate logic, represented in the model by the function `checkSpecial`. We shall limit our attention here to the more interesting case of two numerical operands with a nonzero sum.

17.1 Parameters and Input Assumptions

The inputs of `fadd64` include the following:

- `ui64 opa`: A double-precision encoding of the first operand.
- `ui117 opp`: A 117-bit representation of the second operand. In the FMA case, as seen in Chap. 16, the multiplier produces a 106-bit product, and therefore, the width of the mantissa field of this operand is 105 rather than 52. In the FADD case, the most significant 64 bits form a DP encoding of the operand and the low 53 bits are 0. The name of this operand is intended to suggest its relation

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_17

to the product in the FMA case and to avoid name conflict with the operands of `fmul64`.
- `bool fz, dn`: The FZ and DN fields of the FPSCR (Sect. 14.1).
- `ui2 rmode`: The RC field of the FPSCR, a 2-bit encoding of an IEEE rounding mode (Table 14.1), which we shall denote as $\mathcal{R}$.
- `bool fma`: An indication of an FMA operation.

The remaining inputs are generated by the multiplier in the FMA case and are ignored in the FADD case:

- `bool expOvfl`: An indication that the exponent of the product exceeds the normal 11-bit range.
- `ui8 mulExcp`: A vector representing the exceptional conditions reported by the multiplier, corresponding to $FPSCR[7:0]$. This is the *flags* output of *fmul64*. We use the mnemonics defined in Fig. 14.1 to refer to the bits of this vector. Recall that the inexact indication $mulExcp[IXC]$ does not indicate an exception, but rather that the product is subnormal, the 105-bit mantissa field of `opp` has been right-shifted in order to produce a zero exponent field, and at least one nonzero bit has been shifted out.
- `bool inz`: An indication that the multiplier output is an infinity, a NaN, or a zero.
- `bool piz`: An indication that the multiplier operands were an infinity and a zero.

Two results are returned by `fadd64`:

- `ui64 D`: The double-precision data result.
- `ui8 flags`: The exception flags.

The local variables include the following:

- The sign, exponent, and mantissa fields of the operands, with the mantissa of *opa* zero-extended to 105 bits to match that of *opp*; the mantissa of *opa* is coerced to 0 if the exponent field is 0 and $fz = 1$, and the same holds for *opp* if $fma = 0$:

$$signa = opa[63],\; signp = opp[116]$$
$$expa = opa[62:52],\; expp = opp[115:105]$$
$$fraca = \begin{cases} 0 & \text{if } expa = 0 \text{ and } fz = 1 \\ 2^{53} \cdot opa[51:0] & \text{otherwise} \end{cases}$$
$$fracp = \begin{cases} 0 & \text{if } expp = 0, fz = 1, \text{ and } fma = 0 \\ opp[104:0] & \text{otherwise} \end{cases}$$

- The significand of each operand, formed by appending a zero bit to the mantissa and prepending an integer bit:

$$siga = \begin{cases} 2^{106} + 2 \cdot fraca & \text{if } expa > 0 \\ 2 \cdot fraca & \text{if } expa = 0 \end{cases}$$

$$sigp = \begin{cases} 2^{106} + 2 \cdot fracp \text{ if } expp > 0 \\ 2 \cdot fracp \text{ if } expp = 0 \end{cases}$$

The reason for appending the zero is to avoid loss of accuracy in the near case in the event of a 1-bit right shift (see the proof of Lemma 17.7).

- Qualified versions of $mulExcps[IXC]$ and $expOvfl$:

$$mulStk = \begin{cases} mulExcps[IXC] & \text{if } fma = 1 \\ 0 & \text{if } fma = 0, \end{cases}$$

$$mulOvfl = \begin{cases} expOvfl & \text{if } fma = 1 \text{ and } inz = 0 \\ 0 & \text{otherwise.} \end{cases}$$

The design implicitly utilizes a floating-point format with an 11-bit biased exponent and a 107-bit significand with explicit integer bit. For the purpose of decoding values represented in this format, given bit vectors b, e, and s (representing sign, exponent, and significand, respectively), we define

$$\delta(e, s) = \begin{cases} 2^{e-(2^{10}-1)-106} s & \text{if } e > 0 \\ 2^{1-(2^{10}-1)-106} s & \text{if } e = 0 \end{cases}$$

and

$$\Delta(b, e, s) = \begin{cases} \delta(e, s) & \text{if } b = 0 \\ -\delta(e, s) & \text{if } b \neq 0. \end{cases}$$

In the event that opa is a numerical encoding, i.e., $expa < 2^{11} - 1$, we define A to be its value, possibly forced to zero:

$$A = \Delta(signa, expa, siga) = \begin{cases} 0 & \text{if } fz = 1 \text{ and } expa = 0 \\ decode(opa, DP) & \text{otherwise.} \end{cases}$$

P will denote the value represented by the second operand. In the FADD case, if $opp[116:53]$ is a numerical encoding, the we define P by the same formula as A:

$$\begin{aligned} P &= \Delta(signp, expp, sigp) \\ &= \begin{cases} 0 & \text{if } fz = 1 \text{ and } expp = 0 \\ decode(opp[116:53], DP) & \text{otherwise.} \end{cases} \end{aligned}$$

In the FMA case, we assume that opp and related inputs are supplied by the multiplier, i.e.,

$$\langle opp, mulExcps, piz, inz, mulOvfl \rangle = fmul64(opb, opc, fz, dn, rmode, fma),$$

where *opb* and *opc* are *DP* encodings. In the event that both are numerical, we define

$$B = decode(opb, DP),$$

$$C = decode(opc, DP),$$

and

$$P = BC.$$

The following properties hold in general:

Lemma 17.1 *Assume that if* $fma = 0$*, then* $opp[116 : 53]$ *is numerical, and if* $fma = 1$*, then opb and opc are numerical.*

(a) If $P \neq 0$*, then* $signp = 1 \Leftrightarrow P < 0$.
(b) If $mulOvfl = 1$*, then*

$$|P| \geq 2^{2^{10}+1};$$
$$mulStk = 0.$$

(c) If $mulOvfl = 0$ *and* $expp > 0$*, then*

$$P = \Delta(signp, expp, sigp);$$
$$mulStk = 0.$$

(d) If $mulOvfl = 0$ *and* $expp = 0$*, then*

$$\delta(expp, sigp^{(-53)}) \leq |P| < \delta(expp, sigp^{(-53)} + 2^{53});$$
$$\delta(expp, sigp^{(-53)}) = |P| \Leftrightarrow sigp[52 : 0] = mulStk = 0;$$

Proof In the case $fma = 0$, (a) holds trivially and

$$mulOvfl = mulStk = opp[52 : 0] = 0.$$

Thus, (b) holds vacuously and (c) is true by definition. If $expp = 0$, then by Lemma 2.15,

$$sigp[52 : 0] = (2fracp)[52 : 0] = 2fracp[51 : 0] = 2opp[51 : 0] = 0,$$

and by Definition 1.5 and Lemma 4.15,

$$sigp^{(-53)} = 2^{53} \left\lfloor \frac{sigp}{2^{53}} \right\rfloor = sigp.$$

Thus, (d) reduces to $P = \delta(expp, sigp)$, which again holds by definition.

In the case $fma = 1$, we invoke Lemmas 16.11 and 16.12, substituting *opb*, *opc*, *opp*, *mulExcps*, and P for *opa*, *opb*, D, *flags*, and AB, respectively. First suppose

$P = 0$. Lemma 16.12 yields

$$mulOvfl = expOvfl = 0,$$

$$mulStk = mulExcps[IXC] = 0,$$

$$opp[52:0] = 0,$$

and by Lemma 16.12(c)(v), $opp[115:53] = 0$. It follows that

$$expp = sigp = P = 0,$$

and (c) holds trivially.

If $P \neq 0$, then Lemma 16.11 applies and (a) and (b) follow immediately. If $expp > 0$, then by Lemma 16.11(e),

$$\begin{aligned}
\delta(expp, sigp) &= 2^{expp-(2^{10}-1)-106} sigp \\
&= 2^{opp[115:105-(2^{10}-1)-106}(2^{106} + 2opp[104:0]) \\
&= 2^{opp[115:105-(2^{10}-1)}(1 + 2^{-105} opp[104:0]) \\
&= |P|,
\end{aligned}$$

and (c) follows. If $expp = 0$, then

$$sigp^{(-53)} = 2^{53} \left\lfloor \frac{sigp}{2^{53}} \right\rfloor = 2^{53} \left\lfloor \frac{opp[104:0]}{2^{53}} \right\rfloor = 2^{53} opp[104:52],$$

and (d) similarly follows from Lemma 16.11(f). □

We shall focus on the case of numerical operands with a nonzero sum. This is precisely the complement of the case in which the function *checkSpecial* sets the variable *isSpecial*:

Lemma 17.2 *isSpecial* $= 1 \Leftrightarrow$ *any of the following conditions holds:*

(a) $fma = 0$ *and either opa or* $opp[116:53]$ *is non-numerical;*
(b) $fma = 1$ *and either opa, opb, or opc is non-numerical;*
(c) $A + P = 0$.

Proof It is clear by inspection of the definitions of *opaz* and the local variables of *checkSpecial* that *opa* is non-numerical iff any of the variables *opaInf*, *opaQnan*, and *opaSnan* is set, and that $A = 0$ iff *opaZero* is set. In the case $fma = 0$, $opp[116:53]$ and P are analogously related to *oppInf*, *oppQnan*, *oppSnan*, and *oppZero*. In the case $fma = 1$, by examining the same definitions and Lemma 16.12, it is easily seen that at least one of *opb* and *opc* is non-numerical iff any of the variables *oppInf*, *oppQnan*, and *oppSnan* is set, and again that $P = 0$ iff *oppZero* is set.

Thus, we may assume that all operands are numerical and that A and P are not both 0, and we must show that $A + P = 0$ iff the following conditions hold: $expa = expp$, $fraca = fracp$, $signa \neq signp$, and $mulovfl = mulStk = 0$. We shall derive this equivalence from Lemma 17.1

By Lemma 17.1(a) and (b), we may assume that $signa \neq signp$ and $mulOvfl = 0$. If $expp > 0$, then by Lemma 17.1(c), $mulStk = 0$ and

$$A + P = 0 \Leftrightarrow \delta(expa, siga) = \delta(expp, sigp),$$

and it is easily shown that the latter equation holds iff $expa = expp$ and $fraca = fracp$. Thus, we may assume that $expp = 0$ and appeal to Lemma 17.1(d).

If $sigp[52 : 0] = mulStk = 0$, then

$$|P| = \delta(expp, sigp^{(-53)}) = \delta(expp, sigp)$$

and the claim follows as in the case $expp > 0$. In the remaining case, either $mulStk = 1$ or $fraca \neq fracp$, and

$$\delta(0, sigp^{(-53)}) < |P| < \delta(expp, sigp^{(-53)} + 2^{53}),$$

where

$$\delta(0, sigp^{(-53)}) = 2^{1-(2^{10}-1)-106}2^{53}\left\lfloor \frac{sigp}{2^{53}} \right\rfloor = 2^{-52-(2^{10}-1)}\left\lfloor \frac{sigp}{2^{53}} \right\rfloor$$

and

$$\delta(0, sigp^{(-53)} + 2^{53}) = 2^{-52-(2^{10}-1)}\left(\left\lfloor \frac{sigp}{2^{53}} \right\rfloor + 1\right).$$

This implies $2^{52+(2^{10}-1)}|P| \notin \mathbb{Z}$. On the other hand,

$$2^{52+(2^{10}-1)}|A| = 2^{52+(2^{10}-1)}\delta(0, siga) = 2^{-53}siga = 2opa[51 : 0] \in \mathbb{Z},$$

and hence $A + P \neq 0$. □

The analysis to follow, through Lemma 17.26 of Sect. 17.7, will be based on the input conditions prescribed above along with the additional assumption that $isSpecial = 0$. In the lemmas of Sects. 17.2–17.6, we shall further assume that $mulOvfl = 0$. The case $mulOvfl = 1$ will be handled specially in the proof of the main result of Sect. 17.7.

17.2 Alignment

Prior to the addition, the significands are aligned by a right shift of the significand corresponding to the lesser exponent. The design follows the convention of employing two parallel data paths, *near* and *far*. The near path is used when the signs are opposite and the exponents differ by at most 1. This is the condition under which massive cancellation may occur.

Lemma 17.3 $\mathit{near} = 1 \Leftrightarrow \mathit{signa} \neq \mathit{signp}$ *and* $|\mathit{expa} - \mathit{expp}| \leq 1$.

Proof Since

$$\mathit{exppP1} = (\mathit{expp} + 1) \bmod 2^{12} = \mathit{expp} + 1,$$

$$\mathit{expaEQexppP1} = 1 \Leftrightarrow \mathit{expa} = \mathit{expp} + 1.$$

Similarly

$$\mathit{exppEQexpaP1} = 1 \Leftrightarrow \mathit{expp} = \mathit{expa} + 1,$$

and the claim follows. □

By definition, *signl* and *expl* are the sign and exponent fields of the larger operand. We shall also define *signs* and *exps* to be the corresponding fields of the smaller. This selection is based on the variable *oppGEopa*, which requires justification:

Lemma 17.4 $\mathit{oppGEopa} = 1 \Leftrightarrow |P| \geq |A|$.

Proof If $\mathit{expa} \leq \mathit{expp}$ and $\mathit{siga} \leq \mathit{sigp}$, then $\mathit{oppGEopa} = 1$ and

$$\mathit{siga} = \mathit{siga}^{(-53)} \leq \mathit{sigp}^{(-53)},$$

which implies

$$|A| = \delta(\mathit{expa}, \mathit{siga}) \leq \delta(\mathit{expa}, \mathit{sigp}^{(-53)}) \leq \delta(\mathit{expp}, \mathit{sigp}^{(-53)}) \leq |P|.$$

If $\mathit{expp} \leq \mathit{expa}$ and $\mathit{sigp} < \mathit{siga}$, then $\mathit{oppGEopa} = 0$, and since

$$\mathit{sigp}^{(-53)} < \mathit{siga} = \mathit{siga}^{(-53)},$$

$$\mathit{sigp}^{(-53)} \leq \mathit{siga} - 2^{53},$$

and

$$|P| < \delta(\mathit{expp}, \mathit{sigp}^{(-53)} + 2^{53}) \leq \delta(\mathit{expp}, \mathit{siga}) \leq \delta(\mathit{expa}, \mathit{siga}) = |A|.$$

In the remaining case, *expa* and *expp* are positive and distinct. If $expa > expp$, then $oppGEopa = 0$ and

$$\begin{aligned}|P| &= \delta(expp, sigp)\\ &< \delta(expp, 2^{107})\\ &\leq \delta(expa-1, 2^{107})\\ &= \delta(expa, 2^{106})\\ &\leq \delta(expa, siga)\\ &= |A|,\end{aligned}$$

and similarly, if $expa < expp$, then $oppGEopa = 1$ and $|A| < |P|$. □

As an immediate consequence, *signl* represents the sign of the sum:

Lemma 17.5 $signl = \begin{cases} 0 \text{ } \textit{if} \, |A+P| > 0 \\ 1 \text{ } \textit{if} \, |A+P| < 0. \end{cases}$

In the far case with opposite signs, both significands are shifted left by 1 bit to facilitate rounding, and the exponent must be adjusted accordingly. We define

$$expl' = \begin{cases} expl - 1 & \text{if } far = 1 \text{ and } signa \neq signp \\ expl & \text{otherwise.} \end{cases}$$

The resulting significands are *sigaPrime* and *sigpPrime*, which we shall denote as $siga'$ and $sigp'$. The one that corresponds to the larger operand is the value of *sigl*. The other is the value of *sigs*, which is shifted right to produce *sigShft*:

Lemma 17.6

(a) $sigShft = \lfloor 2^{-expDiff} sigs \rfloor$;
(b) $shiftOut = 1 \Leftrightarrow sigShft \neq 2^{-expDiff} sigs$.

Proof If $expDiff < 128$, then $rshift = expDiff$ and the lemma is trivial. If $expDiff \geq 128$, then $rshift \geq 112$,

$$sigShft = \lfloor 2^{-rshift} sigs \rfloor = 0 = \lfloor 2^{-expDiff} sigs \rfloor,$$

and

$$shiftOut = 1 \Leftrightarrow sigs \neq 0 \Leftrightarrow sigShft \neq 2^{-expDiff} sigs.$$

□

Lemma 17.7 *near* $= 1 \Rightarrow$ *shiftOut* $= 0$.

Proof By Lemma 17.3, $expdiff \in \{0, 1\}$. Since $sigs[0] = 0$, *sigs* is even and $2^{-expDiff} sigs \in \mathbb{Z}$. The claim follows from Lemma 17.6. □

Lemma 17.8 $shiftOut = 1 \Rightarrow sigShft + 2^{1-expDiff} \leq 2^{-expDiff} sigs \leq sigShft + 1 - 2^{1-expDiff}$.

Proof Let $k = sigs \bmod 2^{expDiff} \neq 0$. Then

$$sigs = \lfloor 2^{-expDiff} sigs \rfloor 2^{expDiff} + k = sigShft \cdot 2^{expDiff} + k,$$

or

$$2^{-expDiff} sigs = sigShft + 2^{-expDiff} k.$$

Since *sigs* is even, k is even, and hence $2 \leq k \leq 2^{expDiff} - 2$, or

$$2^{1-expDiff} \leq 2^{-expDiff} k \leq 1 - 2^{1-expDiff},$$

and the lemma follows. □

17.3 Addition

Next we establish error bounds for the sum. We first consider the case in which there is no error in the approximation of the product:

Lemma 17.9 *Assume that either* $expp > 0$ *or* $mulStk = sigp[52:0] = 0$.

(a) $\delta(expl', sum) \leq |A + P| < \delta(expl', sum + 1)$;
(b) $\delta(expl', sum) = |A + P| \Leftrightarrow stk = 0$.

Proof First suppose $signa = signp$. Then $usa = 0$ and $sum = sigl + sigShft$, and

$$\begin{aligned} |A + P| &= \delta(expl', sigl) + \delta(expl', 2^{-expDiff} sigs) \\ &= \delta(expl', sigl + 2^{-expDiff} sigs). \end{aligned}$$

If $shiftOut = 0$, then $stk = mulStk = 0$ and by Lemma 17.6, $\delta(expl', sum) = |A + P|$.

On the other hand, if $shiftOut = 1$, then $stk = 1$ and by Lemma 17.8,

$$sum < sigl + 2^{-expDiff} sigs \leq sum + 1 - 2^{1-expDiff},$$

which implies

$$\delta(expl', sum) < |A + P| \leq \delta(expl', sum + 1 - 2^{1-expDiff}) < \delta(expl', sum + 1).$$

In the remaining case, $signa \neq signp$ and

$$|A + P| = \delta(expl', sigl - 2^{-expDiff} sigs).$$

Suppose $shiftOut = 1$. Then $stk = 1$, $far = 1$, $cin = 0$, and

$$sum = sigl - sigShft - 1.$$

By Lemma 17.8,

$$sum + 2^{1-expDiff} \leq sigl - 2^{-expDiff} sigs \leq sum + 1 - 2^{1-expDiff}.$$

Thus,

$$\delta(expl', sum) + \delta(expl', 2^{1-expDiff}) \leq |A + P| \leq \delta(expl', sum + 1) - \delta(expl', 2^{1-expDiff})$$

and the lemma follows trivially.

But if $shiftOut = 0$, then $stk = 0$ and $cin = 1$. By Lemma 17.6,

$$sum = sigl - sigShft = sigl - 2^{-expDiff} sigs,$$

and hence

$$\delta(expl', sum) = \delta(expl', sigl - 2^{-expDiff} sigs) = |A + P|.$$

□

In the case of an approximation error, there is a loss of precision of the sum:

Lemma 17.10 *Assume that* $expp = 0$ *and either* $mulStk = 1$ *or* $sigp[52 : 0] \neq 0$.

(a) $\delta(expl', sum^{(-53)}) < |A + P| < \delta(expl', sum^{(-53)} + 2^{53})$*;*
(b) Either $stk = 1$ *or* $sum[52 : 0] \neq 0$.

Proof Suppose first that $oppGEopa = 1$. Then $expa = expp = expp' = expl = expl' = 0$, $sigl = sigp' = sigp$, $sigs = siga' = siga$, and $shiftOut = 0$.

If $signa = signp$, then $|A + P| = |A| + |P|$ and $sum = siga + sigp$. Since $siga[53 : 0] = 0$,

$$\begin{aligned}\delta(expl', sum^{(-53)}) &= \delta(0, siga) + \delta(0, sigp^{(-53)}) \\ &< |A| + |P| \\ &< \delta(0, siga) + \delta(0, sigp^{(-53)} + 2^{53})\end{aligned}$$

$$= \delta(expl', sum^{(-53)} + 2^{53}).$$

On the other hand, if $signa \neq signp$, then $sum = sigp - siga$, $sum^{(-53)} = sigp^{(-53)} - siga$ and the same results follow similarly.

Thus, $sum = sigp \pm siga$. If $stk = 0$, then $mulStk = 0$, $sigp[52:0] \neq 0$ and $siga[52:0] = 0$, and it follows that $sum[52:0] \neq 0$.

Now suppose $oppGEopa = 0$. In this case, $sigl = siga'$ $sigs = sigp'$, and $sigShft = \lfloor 2^{-expDiff} sigp' \rfloor$.

If $signa = signp$, then $|A + P| = |A| + |P|$, $expl' = expl = expa$, $siga' = siga$, $sigp' = sigp$, and $sum = siga + \lfloor 2^{-expDiff} sigp \rfloor$. We invoke Lemma 1.30 with the substitutions $k = 53$, $n = expDiff$, and $x = sigp$. Part (a) of the lemma yields

$$\lfloor 2^{-expDiff} sigp \rfloor^{(-53)} = \left\lfloor \frac{x}{2^k} \right\rfloor^{(-n)} \leq \frac{x^{(-n)}}{2^k} = 2^{-expDiff} sigp^{(-53)},$$

which implies

$$\begin{aligned}
\delta(expl', sum^{(-53)}) &= \delta(expa, siga) + \delta(expa, \lfloor 2^{-expDiff} sigp \rfloor^{(-53)}) \\
&\leq \delta(expa, siga) + \delta(expa, 2^{-expDiff} sigp^{(-53)}) \\
&= |A| + \delta(0, sigp^{(-53)}) \\
&< |A| + |P|.
\end{aligned}$$

Part (b) yields

$$\begin{aligned}
2^{-expDiff}\left(sigp^{(-53)} + 2^{53}\right) &= \frac{x^{\langle n \rangle} + 2^n}{2^k} \\
&\leq \left\lfloor \frac{x}{2^k} \right\rfloor^{\langle n \rangle} + 2^n \\
&= \lfloor 2^{-expDiff} sigp \rfloor^{(-53)} + 2^{53},
\end{aligned}$$

which implies

$$\begin{aligned}
\delta(expl', sum^{\langle n \rangle} + 2^{53}) &= \delta(expa, siga) + \delta(expa, \lfloor 2^{-expDiff} sigp \rfloor^{(-53)} + 2^{53}) \\
&\geq \delta(expa, siga) + \delta(expa, 2^{-expDiff}(sigp^{(-53)} + 2^{53})) \\
&= |A| + \delta(0, sigp^{(-53)} + 2^{53}) \\
&> |A| + |P|.
\end{aligned}$$

On the other hand, suppose $signa \neq signp$. Then $|A + P| = |A| - |P|$.

Let $x = 2^{106+(2^{10}-1)-1}|P|$, so that $\delta(0, x) = |P|$, and let

$$n = \begin{cases} expa - 2 & \text{if } expa > 1 \\ 0 & \text{if } expa \leq 1. \end{cases}$$

A straightforward case analysis shows that

$$2^{-expDiff} sigp' = 2^{-n} sigp$$

and

$$\delta(expl', 2^{-n} x) = \delta(0, x).$$

Since

$$\delta(0, sigp^{(-53)}) < \delta(0, x) < \delta(0, sigp^{(-53)} + 2^{53}),$$

$$2^{53} \left\lfloor \frac{sigp}{2^{53}} \right\rfloor = sigp^{(-53)} < x < sigp^{(-53)} + 2^{53} = 2^{53} \left(\left\lfloor \frac{sigp}{2^{53}} \right\rfloor + 1 \right)$$

and

$$\left\lfloor \frac{sigp}{2^{53}} \right\rfloor < \frac{x}{2^{53}} < \left\lfloor \frac{sigp}{2^{53}} \right\rfloor + 1,$$

which implies $2^{-53} x \notin \mathbb{Z}$ and $\lfloor 2^{-53} x \rfloor = \lfloor 2^{-53} sigp \rfloor$.

We shall show that

$$sum^{(-53)} = (siga' - 2^{-n} x)^{(-53)}.$$

If $cin = 1$, then $shiftOut = mulStk = 0$. It follows that $sigp[52 : 0] \neq 0$, i.e., $2^{-53} sigp \notin \mathbb{Z}$. Consequently,

$$\left\lfloor -\frac{sigp}{2^{n+53}} \right\rfloor = -\left\lfloor \frac{sigp}{2^{n+53}} \right\rfloor - 1 = -\left\lfloor \frac{x}{2^{n+53}} \right\rfloor - 1 = \left\lfloor -\frac{x}{2^{n+53}} \right\rfloor,$$

which implies $(-2^{-n} sigp)^{(-53)} = (-2^{-n} x)^{(-53)}$. Since $shiftOut = 0$,

$$sum = siga' - \lfloor 2^{-expDiff} sigp' \rfloor = siga' - 2^{-expDiff} sigp' = siga' - 2^{-n} sigp,$$

and

$$\begin{aligned} sum^{(-53)} &= (siga' - 2^{-n} sigp)^{(-53)} \\ &= siga' + (-2^{-n} sigp)^{(-53)} \\ &= siga' + (-2^{-n} x)^{(-53)} \end{aligned}$$

$$= (siga' - 2^{-n}x)^{(-53)}.$$

For the case $cin = 0$, we invoke Lemma 1.31 to n and x along with $y = sigp$ and $k = 53$. This yields

$$(-\lfloor 2^{-n} sigp \rfloor - 1)^{(-53)} = (-2^{-n}x)^{(-53)}.$$

Thus, if $cin = 0$, then

$$sum = siga' - \lfloor 2^{-expDiff} sigp' \rfloor - 1 = siga' - \lfloor 2^{-n} sigp \rfloor - 1$$

and

$$\begin{aligned} sum^{(-53)} &= siga'^{(-53)} + (-\lfloor 2^{-n} sigp \rfloor - 1)^{(-53)} \\ &= siga' + (-2^{-n}x)^{(-53)} \\ &= (siga' - 2^{-n}x)^{(-53)}, \end{aligned}$$

as in the case $cin = 1$.

This equation is equivalent to

$$sum^{(-53)} \leq siga' - 2^{-n}x < sum^{(-53)} + 2^{53}.$$

In fact, both inequalities are strict, for otherwise

$$2^{-n}x = siga' - sum^{(-53)}$$

and

$$\frac{x}{2^{53}} = 2^n \left(\frac{siga'}{2^{53}} - \left\lfloor \frac{sum}{2^{53}} \right\rfloor \right) \in \mathbb{Z}.$$

Thus,

$$\delta(expl', sum^{(-53)}) < \delta(expl', siga') - \delta(expl', 2^{-n}x) < \delta(expl', sum^{(-53)} + 2^{53}),$$

where

$$\delta(expl', siga') = \delta(expa, siga) = |A|$$

and

$$\delta(expl', 2^{-n}x) = \delta(0, x) = |P|.$$

Thus, if $stk = 0$, then $mulStk = shiftOut = 0$ and $sum = siga' \pm sigShft$, where $sigShft = 2^{-n} sigp$ for some $n \geq 0$. Since $siga'[52:0] = 0$ and

$$(2^{-n} sigp)[52:0] = 2^{-n} sigp[52+n:0] \neq 0,$$

$sum[52:0] \neq 0$. □

17.4 Leading Zero Anticipation

The near case may involve cancellation, requiring a normalizing left shift of the sum. The exponent of the sum is estimated in advance of the addition by the method of leading zero anticipation described in Sect. 8.2. Given two 128-bit vectors, the function *LZA128* computes a vector with an exponent that is either equal to or one less than that of the 128-bit sum and passes that vector to the leading zero counter *CLZ128*. This is a 128-bit version of the function *CLZ53*, which we analyzed in Sect. 16.3.

Lemma 17.11 *If x is a 128-bit vector and $x \neq 0$, then*

$$CLZ128(x) = 127 - expo(x).$$

Proof The proof is essentially the same as that of Lemma 16.6. □

Lemma 17.12 *If a and b are 128-bit vectors with $s = a + b > 2^{128}$, then*

$$127 - LZA128(a, b) \leq expo(s[127:0]) \leq 128 - LZA128(a, b).$$

Proof Let p, k, and w be as defined in the body of *LZA128*. Applying Lemma 8.6 with $n = 128$, we have

$$0 \leq expo(w) - 1 \leq expo(s[127:0]) \leq expo(w),$$

and the claim follows from Lemma 17.11. □

We shall also require the following two basic properties of *LZA128*.

Lemma 17.13 *If a and b are 128-bit vectors, then $LZA128(a, b) > 0$.*

Proof According to the definition of *LZA128*,

$$LZA128(a, b) = CLZ128\left(\left\lfloor \frac{w}{2} \right\rfloor\right),$$

where w is a 128-bit vector. By Lemma 17.11,

$$CLZ128\left(\left\lfloor \frac{w}{2} \right\rfloor\right) = 127 - expo\left(\left\lfloor \frac{w}{2} \right\rfloor\right) \geq 127 - 126 = 1.$$

□

Lemma 17.14 *Let a, b, a', and b' be 128-bit vectors and let $\ell = LZA128(a, b)$. If $a'[i] = a[i]$ and $b'[i] = b[i]$ for all $i \geq 127 - \ell$, then $LZA128(a', b') = \ell$.*

Proof Let w be as defined in the body of *LZA128* and let $x = \left\lfloor \frac{w}{2} \right\rfloor$. Let w' and x' be the corresponding values for a' and b'. By Lemma 17.11, $expo(x) = 127 - \ell$. Thus, $a'[i] = a[i]$ and $b'[i] = b[i]$ for all $i \geq expo(x)$. It follows that $w'[i] = w[i]$ for all $i \geq expo(x) + 1$, and $x'[i] = x[i]$ for all $i \geq expo(x)$, which implies $expo(x') = expo(x)$, and hence $LZA128(a', b') = 127 - expo(x') = \ell$. □

We apply these results to the output of the adder:

Lemma 17.15 *If $near = 1$ and $exps \neq 0$, then*

$$106 - lza \leq expo(sum) \leq 107 - lza.$$

Proof The hypothesis implies that

$$rshift = expDiff = |expa - expp| \in \{0, 1\}$$

and

$$sigShft = 2^{-rshift} sigs = \begin{cases} sigs \text{ if } expa[0] = expp[0] \\ \frac{1}{2} sigs \text{ if } expa[0] \neq expp[0]. \end{cases}$$

It also implies $stk = 0$, and therefore $\delta(expl', sum) = |A + P| \neq 0$, which implies $sum \neq 0$. Furthermore, $cin = 1$, which implies $sum = sigl - sigShft$.

Let $s = in1LZA + in2LZA$. By inspection, $in1LZA = 2^{21} sigl$, $in2LZA = 2^{128} - 2^{21} sigShft - 1 = ops$, and

$$s = 2^{128} + 2^{21}(sigl - sigShft) - 1 = 2^{128} + 2^{21} sum - 1.$$

Let $in1LZA' = in1LZA + 1$, $lza' = LZA128(in1LZA', in2LZA)$, and $s' = in1LZA' + in2LZA = 2^{128} + 2^{21} sum$. Then $s' > 2^{128}$, $s'[127:0] = s' \bmod 2^{128} = 2^{21} sum$ and by Lemma 17.12,

$$127 - lza' \leq expo(2^{21} sum) \leq 128 - lza',$$

or

$$106 - lza' \leq expo(sum) \leq 107 - lza',$$

and we need only show $lza' = lza$. But since $in1LZA[0] = 0$, $in1LZA'[i] = in1LZA[i]$ for all $i > 0$ and since

$$127 - lza' = (107 - lza') + 10 \geq expo(sum) > 0,$$

this follows from Lemma 17.14. □

17.5 Normalization

In the near case, if the anticipated number of leading zeroes is less than the exponent of the larger operand, then it becomes the shift amount and the exponent is adjusted accordingly. Otherwise, the shift is limited by the exponent, which is then reduced to 0. We have estimates for the exponent of the shifted sum in both cases. The estimates also apply to the far case.

Lemma 17.16

(a) $sumShft = 2^{lshift} sum$*;*
(b) $expShft > 0 \Rightarrow expo(sumShft) \geq 106$*;*
(c) $expShft = 0 \Rightarrow expo(sumShft) \leq 106$.

Proof

(a) Since $sumShft = (2^{lshift} sum) \bmod 2^{108}$, we need only show that $2^{lshift} sum < 2^{108}$, i.e., $lshift + expo(sum) < 108$. This holds trivially if $far = 1$, since $lshift = 0$. If $far = 0$, then $lshift \leq lza$ and Lemma 17.15 applies.

(b) First suppose $far = 1$. If $signa = signp$, then

$$sumShft = sum = sigl + sigShft \geq sigl \geq 2^{106}$$

and

$$expo(sumShft) \geq 106.$$

But if $signa \neq signp$, then

$$sumShft = sum = sigl - sigShft - 1 + cin,$$

where $sigl \geq 2^{107}$ and $sigShft = \lfloor 2^{-expDiff} sigs \rfloor$. If $exps > 0$, then $expDiff = expl - exps \geq 2$, $sigs < 2^{108}$, and $sigShft < 2^{106}$. If $exps = 0$, then $expDiff = expl - exps - 1 \geq 1$, $sigs < 2^{107}$, and again $sigShft < 2^{106}$. Thus,

$$sumShft > 2^{107} - 2^{106} - 1 = 2^{106} - 1,$$

i.e., $sumShft \geq 2^{106}$.

In the remaining case, $far = 0$ and $lshft = lza < expl$. By Lemma 17.13, $expl > 1$, which implies $exps > 0$, and Lemma 17.12 yields

$$expo(sumShft) = lshift + expo(sum) = lza + expo(sum) \geq 106.$$

(c) If $far = 1$, then $expShft = 0$ implies $usa = expl = 0$ and

$$sumShft = sum = sigs + sigl < 2^{106} + 2^{106} = 2^{107}.$$

Thus, we may assume $far = 0$, and hence $lza \geq expl$. If $expl \leq 1$, then $lshift = 0$ and

$$sumShft = sum \leq sigl < 2^{107}.$$

But if $expl > 1$, then $exps > 0$, $lshift = expl - 1 \leq lza - 1$, and by Lemma 17.12,

$$expo(sumShft) = lshift + expo(sum) \leq lza - 1 + expo(sum) \leq 106.$$

□

The accuracy of the sum, as given by Lemmas 17.9 and 17.10, is preserved by the normalization.

Lemma 17.17 *Assume that either* $expp > 0$ *or* $mulStk = sigp[52:0] = 0$.

(a) $\delta(expShft, sumShft) \leq |A + P| < \delta(expShft, sumShft + 1)$*;*
(b) $\delta(expShft, sumShft) = |A + P| \Leftrightarrow stk = 0$.

Proof If $far = 1$, then $expShft = expl'$, $sumShft = sum$, and the lemma follows from Lemma 17.9. Thus, we may assume $far = 0$, which implies $expl' = expl$, and the lemma will follow from Lemma 17.9 once we prove the following two claims:

(1) $\delta(expShft, sumShft) = \delta(expl, sum)$.
(2) If $stk = 1$, then $\delta(expShft, sumShft + 1) = \delta(expl, sum + 1)$.

Note that if $stk = 1$, then since $expDiff \leq 1$ and $sigs[0] = 0$, Lemma 17.6 implies

$$sigShft = \lfloor 2^{-expDiff} sigs \rfloor = 2^{-expDiff} sigs,$$

and hence $shiftOut = 0$, which implies $mulStk = 1$, $expp = 0$, and $expl = expa \leq 1$.

Suppose $lza < expl$. Then $lshift = lza$, $expShft = expl - lza > 0$, $sumShft = 2^{lza} sum$, and

$$\begin{aligned}
\delta(expl, sum) &= 2^{expl-(2^{10}-1)-106} sum \\
&= 2^{expl-lza-(2^{10}-1)-106} 2^{lza} sum \\
&= 2^{expShft-(2^{10}-1)-106} sumShft \\
&= \delta(expShft, sumShft).
\end{aligned}$$

If $stk = 1$, then $lza < expl \leq 1$, which implies = $lshift = lza = 0$, $expShft = expl$, and $sumShft = sum$.

Finally, suppose $lza \geq expl$. Then $expShft = 0$. We may assume $expl > 0$, for otherwise $lshift = 0$, $expShft = expl$, and $sumShft = sum$. Thus, $lshift = expl - 1$ and

$$\delta(expShft, sumShft) = \delta(0, 2^{expl-1} sum)$$

$$= 2^{1-(2^{10}-1)-106} 2^{expl-1} sum$$
$$= 2^{expl-(2^{10}-1)-106} sum$$
$$= \delta(expl, sum).$$

if $stk = 1$, then $lshift = expl - 1 = 0$, which implies $sumShft = sum$ and

$$\delta(expShft, sumShft + 1) = \delta(0, sum + 1) = \delta(1, sum + 1) = \delta(expl, sum + 1).$$

□

The following estimate holds in all cases:

Lemma 17.18

(a) $\delta(expShft, sumShft^{(-53)}) \leq |A + P| < \delta(expShft, sumShft^{(-53)} + 2^{53})$*;*
(b) $\delta(expShft, sumShft^{(-53)}) = |A + P| \Leftrightarrow sumShft[52:0] = stk = 0$.

Proof If $expp > 0$ or $mulStk = sigp[52:0] = 0$, then the lemma is a weakening of Lemma 17.17. In the remaining case, $lshift = 0$, and therefore $sum = sumShft$. If $far = 1$, then $expShft = expl'$; if $far = 0$, then $expl' = expl \leq 1$ and $expShft \leq expl$. In either case, the lemma follows from Lemma 17.10. □

17.6 Rounding

For the purpose of rounding, we distinguish between the *overflow* case, in which the exponent of the shifted sum has the maximum value of 107, and the *normal* case. Rounding is performed for both cases before that exponent is known, and the appropriate result is selected later.

Rounding involves a possible increment of *sumShft* at the index of the least significant bit of the rounded result. For both the overflow and normal cases, we compute a set of masks for extracting the lsb and the guard and sticky bits, which determine whether the increment occurs. These masks are actually applied to the unshifted sum as the shift is being performed.

Lemma 17.19

(a) $lOvfl = sumShft[55]$*;*
(b) $gOvfl = sumShft[54]$*;*
(c) $sOvfl = 0 \Leftrightarrow sumShft[53:0] = stk = 0$*;*
(d) $lNorm = sumShft[54]$*;*
(e) $gNorm = sumShft[53]$*;*
(f) $sNorm = 0 \Leftrightarrow sumShft[52:0] = stk = 0$.

Proof We shall prove (a) and (c); the proofs of the other claims are similar.

(a) If $lshift \leq 55$, then

$$lOvfl = sum \mathbin{\&} lOvflMask = sum \mathbin{\&} 2^{55-lshift} = sum[55-lshift] = sumShft[55].$$

On the other hand, if $lshift > 55$, then

$$lOvfl = 0 = sumShft[55].$$

(c) We shall show that $sum \mathbin{\&} sOvflMask = 0 \Leftrightarrow sumShft[53:0] = 0$. If $lshift \leq 53$, then

$$\begin{aligned} sum \mathbin{\&} sOvflMask = 0 &\Leftrightarrow sum \mathbin{\&} \left\lfloor \frac{2^{54}-1}{2^{lshift}} \right\rfloor = 0 \\ &\Leftrightarrow sum \mathbin{\&} (2^{54-lshift}-1) = 0 \\ &\Leftrightarrow sum[53-lshift:0] = 0 \\ &\Leftrightarrow sumShft[53:0] = 0. \end{aligned}$$

But if $lshift > 53$, then

$$sum \mathbin{\&} sOvflMask = sum \mathbin{\&} 0 = 0 = sumShft[53:0].$$

□

Since the rounder operates on the absolute value of the sum, the rounding mode $\mathcal{R}$ must be adjusted. We define

$$\mathcal{R}' = \begin{cases} RDN \text{ if } \mathcal{R} = RUP \text{ and } signOut = 1 \\ RUP \text{ if } \mathcal{R} = RDN \text{ and } signOut = 1 \\ \mathcal{R} \quad \text{otherwise.} \end{cases}$$

Lemma 17.20

(a) $\mathcal{R}(A+P, 53) = \begin{cases} \mathcal{R}'(|A+P|, 53) \text{ if } signl = 0 \\ -\mathcal{R}'(|A+P|, 53) \text{ if } signl = 1; \end{cases}$

(b) $drnd(A+P, \mathcal{R}, DP) = \begin{cases} drnd(A+P, \mathcal{R}', DP) \text{ if } signl = 0 \\ -drnd(A+P, \mathcal{R}', DP) \text{ if } signl = 1. \end{cases}$

Proof This follows from the above definition, the definition of *signOut*, and Lemma 17.5. □

We begin with the case $expShft > 0$. By Lemma 17.16, $expo(sumShft)$ is either 107 or 106. For both cases, we apply Lemma 6.104 to show that $|A+P|$ is correctly rounded.

Lemma 17.21 *Assume* $expShft > 0$.

(a) If $expo(sumShft) = 107$*, then* $|A + P| \geq spn(DP)$*,*

$$\mathcal{R}'(|A + P|, 53) = 2^{expShft-(2^{10}-1)-51} sumOvfl$$

and

$$\mathcal{R}'(|A + P|, 53) = |A + P| \Leftrightarrow gOvfl = sOvfl = 0.$$

(b) If $expo(sumShft) = 106$*, then* $|A + P| \geq spn(DP)$*,*

$$\mathcal{R}'(|A + P|, 53) = 2^{expShft-(2^{10}-1)-52} sumNorm$$

and

$$\mathcal{R}'(|A + P|, 53) = |A + P| \Leftrightarrow gNorm = sNorm = 0.$$

Proof We instantiate Lemma 6.104 with $n = 53$,

$$z = 2^{53+(2^{10}-1)-expShft}|A + P|,$$

and

$$x = \left\lfloor \frac{sumShft}{2^{53}} \right\rfloor.$$

Upon expanding the definition of δ, Lemma 17.18 yields

$$x \leq z < x + 1$$

and

$$x = z \Leftrightarrow sumShft[52 : 0] = stk = 0.$$

Thus, $x = \lfloor z \rfloor$ and $z \in \mathbb{Z} \Leftrightarrow sumShft[52 : 0] = stk = 0$.

Let $e = expo(x) = expo(RTZ(x, n)) = expo(sumShft) - 53 \in \{53, 54\}$. Then

$$RTZ(x, 53) = 2^{e-52} x[e : e - 52]$$

and by definition,

$$fp^{+}(RTZ(x, 53), 53) = RTZ(x, 53) + 2^{e-52} = 2^{e-52}(x[e : e - 52] + 1).$$

Consider the case $expo(sumShft) = 107$ and $e = 54$. The lower bound on $|A + P|$ follows from Lemma 17.18. According to Lemmas 6.104 (b) and 17.19 and the definition of *incOvfl*,

$$\begin{aligned}\mathcal{R}'(z, 53) &= \begin{cases} RTZ(x, 53) & \text{if } incOvfl = 0 \\ fp^+(RTZ(x, 53), 53) & \text{if } incOvfl = 1 \end{cases} \\ &= 2^{e-52}(x[e : e - 52] + incOvfl) \\ &= 2^2(sumShft[107 : 55] + incOvfl) \\ &= 2^2(sumUnrnd[53 : 1] + incOvfl) \\ &= 2^2 sumOvfl,\end{aligned}$$

or

$$\mathcal{R}'(|A + P|, 53) = 2^{expShft-(2^{10}-1)-51} sumOvfl.$$

Moreover, by Lemmas 6.104 (a) and 17.19,

$$\begin{aligned}&\mathcal{R}'(|A + P|, 53) = |A + P| \\ &\Leftrightarrow z \text{ is 53-exact} \\ &\Leftrightarrow x[1 : 0]]sumShft[54 : 53] = 0 \text{ and } sumShft[52 : 0] = stk = 0 \\ &\Leftrightarrow gOvfl = sOvfl = 0.\end{aligned}$$

The case $e = 106$ is similar, with *incOvfl*, *sumOvfl*, *gOvfl*, and *sOvfl* replaced by *incNorm*, *sumNorm*, *gNorm*, and *sNorm*. □

If $expShft = 0$, then $expo(sumShft) \leq 106$. If $expo(sumShft) = 106$, then the sum has overflowed to the normal range and we again apply Lemma 6.104. Otherwise, the sum is denormal and we apply Lemma 6.108.

Lemma 17.22 *Assume* $expShft = 0$.

(a) If $expo(sumShft) = 106$*, then* $|A + P| \geq spn(DP)$*,*

$$\mathcal{R}'(|A + P|, 53) = 2^{-2^{10}-50} sumNorm,$$

and

$$\mathcal{R}'(|A + P|, 53) = |A + P| \Leftrightarrow gNorm = sNorm = 0.$$

(b) If $expo(sumShft) < 106$*, then* $|A + P| < spn(DP)$*,*

$$drnd(|A + P|, \mathcal{R}', DP) = 2^{-2^{10}-50} sumNorm$$

and

$$drnd(|A+P|, \mathcal{R}', DP) = |A+P| \Leftrightarrow gNorm = sNorm = 0.$$

Proof Following the proof of Lemma 17.21 with *expShft* replaced by 1, let

$$z = 2^{51+2^{10}}|A+P|.$$

Once again, $x = \lfloor 2^{-53} sumShft \rfloor = \lfloor z \rfloor$, $z \in \mathbb{Z} \Leftrightarrow sumShft[52:0] = stk = 0$, and $e = expo(x) = expo(sumShft) - 53$. The proof for the case $expo(sumShft) = 106$ is the same as that of Lemma 17.21. For $expo(sumShft) < 106$, we consider the subcases $expo(sumShft) \geq 54$ and $expo(sumShft) < 54$ separately.

For $expo(sumShft) \geq 54$, we have $e \geq 1$ and we repeat the proof for $expo(sumShft) = 106$ with $n = e$ instead of $n = 53$. This yields

$$\mathcal{R}'(|A+P|, e) = 2^{-2^{10}-50} sumNorm,$$

and

$$\mathcal{R}'(|A+P|, e) = |A+P| \Leftrightarrow gNorm = sNorm = 0.$$

But by definition,

$$drnd(|A+P|, \mathcal{R}, DP) = \mathcal{R}(|A+P|, 53 + expo(|A+P|) - expo(spn(DP))),$$

where

$$53 + expo(|A+P|) - expo(spn(DP)) = 53 + (e - 2^{10} - 51) - (2 - 2^{10}) = e.$$

For the case $expo(sumShft) < 54$, since

$$expo(|A+P|) = e - 2^{10} - 51 < -2^{10} - 50,$$

$$|A+P| < 2^{-2^{10}-50} = spd(DP)$$

and we may invoke Lemma 6.108. Note that since $expo(sumShft) = e + 53 < 54$, $sumUnrnd = sumShft[107:54] = 0$ and $sumNorm = incNorm$. Thus,

$$2^{-2^{10}-50} sumNorm = \begin{cases} spd(DP) & \text{if } incNorm = 1 \\ 0 & \text{if } incNorm = 0. \end{cases}$$

The proof is completed by a straightforward case analysis. Suppose, for example,

$$|A+P| > \frac{1}{2} spd(DP) = 2^{-2^{10}-51}.$$

Then $e = 0$, $expo(sumShft) = 53$, and $gNorm = sumShft[53] = 1$. Furthermore, since

$$|A + P| > 2^{-2^{10}-51} = \delta(0, 2^{53}) = \delta(expShft, sumShft^{(-53)}),$$

Lemma 17.18(b) implies $sNorm = 1$. According to the definition of *incNorm*, it follows that $incNorm = 1 \Leftrightarrow \mathcal{R}' = RNE$ or $\mathcal{R}' = RUP$ and the desired result follows from Lemma 6.108 (c). The cases $|A+P| = \frac{1}{2}spd(DP)$ and $0 < |A+P| < \frac{1}{2}spd(DP)$ are similar. □

In the overflow case, *double overflow* occurs if rounding results in an additional exponent increment. The next three lemmas pertain to the implications of overflow and double overflow, which are indicated by the variables *ovfl* and *ovfl2*.

Lemma 17.23 *If* $expo(sumShft) = 107$, *then*

(a) $ovfl2 = 0 \Rightarrow expo(sumOvfl) = 52$;
(b) $ovfl2 = 1 \Rightarrow sumOvfl = 2^{53}$;
(c) Either $ovfl2 = 1$ *or* $ovfl = 1$.

Proof (a) and (b) are trivial. To prove (c), suppose $ovfl2 = ovfl = 0$. Since $sumUnrnd[53] = sumShft[107] = 1$ and $sumNorm[53] = ovfl = 0$, we must have $sumUnrnd[53:0] = 2^{54} - 1$ and $incNorm = 1$. But then $sumUnrnd[53:1] = 2^{53} - 1$, and since $ovfl2 = 0$, we must also have $incOvfl = 0$.

Suppose $rndDir = rndInf$. Since $incNorm = 1$, either $gNorm = 1$ or $sNorm = 1$. But this implies $sOvfl = 1$, contradicting $incOvfl = 0$.

In the remaining case, $rndDir = rndNear$. Since $incNorm = 1$, $gNorm = 1$, which implies $sOvfl = 1$. Furthermore, $gOvfl = sumShft[54] = sumUnrnd[0] = 1$, again contradicting $incOvfl = 0$. □

Lemma 17.24 *If* $expo(sumShft) = 106$, *then*

(a) $ovfl2 = 0$;
(b) $ovfl = 0 \Rightarrow expo(sumNorm) = 52$;
(c) $ovfl = 1 \Rightarrow sumNorm = 2^{53}$, $sumOvfl = 2^{52}$, $gOvfl = 1$, *and either* $gNorm = 1$ *or* $sNorm = 1$.

Proof (a) and (b) are trivial. To prove (c), suppose $ovfl = sumNorm[53] = 1$. Then $sumUnrnd = 2^{53} - 1$, $incNorm = 1$, and $sumNorm = 2^{53}$. Since $sumUnrnd = 2^{53} - 1$, $gOvfl = sumUnrnd[0] = 1$. Since $incNorm = 1$, *rndDir* is either *rndNear* or *rndInf*, and either $gNorm = 1$ or $sNorm = 1$. It follows that $sOvfl = 1$, which implies $incOvfl = 1$, and hence $sumOvfl = 2^{52}$. □

Lemma 17.25 *If* $expo(sumShft) < 106$, *then*

(a) $ovfl2 = ovfl = 0$;
(b) $sumNorm \leq 2^{52}$.

Proof Both claims are trivial. □

17.7 Correctness Theorems

Our first objective is to relate the function *fadd64* with the specification function *arm-post-comp* of Sect. 14.3, which takes an unrounded value, an initial FPSCR state, and a floating-point format, and returns a rounded result and the updated FPSCR. We shall show that the data value D computed by *fadd64*, which is the concatenation of the fields *signOut*, *expOut*, and *fracOut*, and the post-computation exception flags, $R[IXC]$, $R[UFC]$, and $R[OFC]$, match the corresponding values returned by *arm-post-comp*. The two correctness theorems corresponding to FADD and FMA follow easily from this result:

Lemma 17.26 *Let R_{in} be a 32-bit vector with $R_{in}[12:10] = 0$ and let $fz = R_{in}[24]$, $dn = R_{in}[25]$ and $rmode = R_{in}[23:22]$. Let*

$$\langle D, \mathit{flags}\rangle = \mathit{fadd64}(\mathit{opa}, \mathit{opp}, \mathit{fz}, \mathit{dn}, \mathit{rmode}, \mathit{fma}, \mathit{inz}, \mathit{piz}, \mathit{expOvfl}, \mathit{mulExcps})$$

and

$$\langle D_{spec}, R_{spec}\rangle = \mathit{arm\text{-}post\text{-}comp}(A + P, R_{in}, DP).$$

Then $D = D_{spec}$ and for $k = IXC$, UFC, and OFC, $(\mathit{flags} \mid R_{in})[k] = R_{spec}[k]$.

Proof Note that the two functions *fadd64* and *arm-post-comp* are similarly structured, treating the overflow, underflow, and normal cases in that order.
Case 1: Overflow

The *arm-post-comp* overflow condition is $|\mathcal{R}(A+P, 53)| > lpn(DP)$, which may be expressed as $expo(\mathcal{R}(A + P, 53)) \geq 2^{10}$, and that of *fadd64* is $infOrMax = 1$. Once these conditions are shown to be equivalent, the comparison for this case is trivial.
Subcase 1.1: $mulOvfl = 1$.

In this case, $infOrMax = 1$. Since $expp[11] = 1$, $expp \geq 2^{11}$, which implies $sigp \geq 2^{106}$ and

$$|P| \geq \delta(expp, sigp^{(-53)}) \geq 2^{2^{11}-(2^{10}-1)-106+106} = 2^{2^{10}+1}.$$

On the other hand, since $expa \leq 2^{11} - 2$ and $siga < 2^{107}$,

$$|A| = \delta(expa, siga) < 2^{2^{11}-2-(2^{10}-1)-106+107} = 2^{2^{10}}.$$

Consequently, $|A + P| \geq |P| - |A| > 2^{2^{10}+1} - 2^{2^{10}} = 2^{2^{10}}$ and

$$expo(\mathcal{R}(A + P, 53)) \geq expo(A + P) \geq 2^{10}.$$

Thus, we may assume $mulOvfl = 0$ and Lemmas 17.3–17.25 apply. We may also assume that $expShft > 0$; otherwise, both conditions are clearly false. By Lemma 17.16, $expo(sumShft) \geq 106$.

In order to simplify the definition of *infOrMax*, we note that if $expShft = 2047$, then since

$$expShft \leq expl = \max(expa, expp),$$

our assumptions imply $fma = 1$ and $expp = 2047$, and therefore $opplong = 1$.
Subcase 1.2: $expo(sumShft) = 107$.

If $ovfl2 = 1$, then by Lemmas 17.21 and 17.23,

$$expo(\mathcal{R}(A + P, 53)) = expShft - (2^{10} - 1) - 51 + 53 = expShft - 2^{10} + 3$$

and

$$expo(\mathcal{R}(A + P, 53)) \geq 2^{10} \Leftrightarrow expShft \geq 2^{11} - 3 \Leftrightarrow infOrMax = 1.$$

Similarly, if $ovfl2 = 0$, then $ovfl1 = 1$,

$$expo(\mathcal{R}(A + P, 53)) = expShft - (2^{10} - 1) - 51 + 52 = expShft - 2^{10} + 2,$$

and

$$expo(\mathcal{R}(A + P, 53)) \geq 2^{10} \Leftrightarrow expShft \geq 2^{11} - 2 \Leftrightarrow infOrMax = 1.$$

Subcase 1.3: $expo(sumShft) = 106$.

If $ovfl = 1$, then by Lemmas 17.21 and 17.24,

$$expo(\mathcal{R}(A + P, 53)) = expShft - (2^{10} - 1) - 52 + 53 = expShft - 2^{10} + 2$$

and

$$expo(\mathcal{R}(A + P, 53)) \geq 2^{10} \Leftrightarrow expShft \geq 2^{11} - 2 \Leftrightarrow infOrMax = 1.$$

But if $ovfl = 0$, then

$$expo(\mathcal{R}(A + P, 53)) = expShft - (2^{10} - 1) - 52 + 52 = expShft - 2^{10} + 1$$

and

$$expo(\mathcal{R}(A + P, 53)) \geq 2^{10} \Leftrightarrow expShft \geq 2^{11} - 1 \Leftrightarrow infOrMax = 1.$$

Case 2: Underflow

The *arm-post-comp* underflow condition is $|A + P| < spn(DP)$, or $expo(A + P) < 2 - 2^{10}$, and that of *fadd64* is $tiny = 1$. Once again, we must establish the equivalence of these conditions. By Lemma 17.18

$$expo(A + P) = expo(\delta(expShft, sumShft)),$$

and it follows from Lemma 17.16 that

$$\begin{aligned} expo(A + P) < 2 - 2^{10} &\Leftrightarrow expo(sumShft) < 106 \\ &\Leftrightarrow expo(sumNorm) < 52 \\ &\Leftrightarrow tiny = 1. \end{aligned}$$

If $fz = 1$, then $R[UFC] = 1$, the data result is a 0 with the sign of $A + P$, and the theorem holds trivially. In the remaining case, according to the specification, the sum is rounded to $d = drnd(A + P, \mathcal{R}, DP)$ and if $d \neq A + P$, then $R[UFC] = R[IXC] = 1$. Since the sign of the result is correctly represented by *signl*, we need only show that $|d|$ is correctly encoded by *exp* and *frac*. By Lemma 17.22, $|d| = 2^{-2^{10}-50} sumNorm$.
Subcase 2.1: $sumNorm[52] = 1$.

By Lemma 17.25, $sumNorm = 2^{52}$, which implies $|d| = 2^{2-2^{10}} = spn(DP)$, $expOut = 1$, and $fracOut = 0$. Thus, D is a normal encoding and the encoded absolute value is

$$2^{expOut-(2^{10}-1)}(1 + 2^{-52} fracOut) = 2^{1-(2^{10}-1)} = 2^{2-2^{10}} = |d|.$$

By Lemma 17.22, $|d| = |A + P| \Leftrightarrow gNorm = sNorm = 0$, which is the condition under which $R[UFC]$ and $R[IXC]$ are not set.
Subcase 2.2: $sumNorm[52] = 0$.

In this case, $sumNorm < 2^{52}$, $expOut = 0$, and $fracOut = sumNorm$. Thus, D is a denormal encoding and the encoded absolute value is

$$2^{1-(2^{10}-1)-52} fracOut = 2^{-2^{10}-50} sumNorm = |d|.$$

Correctness of the flags follows as in Subcase 2.1.
Case 3: $A + P$ and $\mathcal{R}(A + P)$ are both within the normal range.

In this case, $expShft \geq 106$, the specified value of D is the normal encoding of $r = \mathcal{R}(A + P, 53)$, and $R[IXC]$ is set when $r \neq A + P$. Again, since it is clear that the sign of the sum is correctly represented by *signl*, we need only show that $|r|$ is correctly encoded by the exponent and mantissa fields, i.e.,

$$|r| = 2^{expOut-(2^{10}-1)}(1 + 2^{-52} fracOut).$$

Subcase 3.1: $ovfl2 = 1$.

In this case, $expOut = expShft + 2$ and $fracOut = 0$. By Lemmas 17.23 and 17.24, $expo(sumShft) = 107$ and $sumOvfl = 2^{53}$. By Lemma 17.21,

$$\begin{aligned}|r| &= 2^{expShft-(2^{10}-1)-51}2^{53} \\ &= 2^{expShft+2-(2^{10}-1)} \\ &= 2^{expOut-(2^{10}-1)}(1+2^{-52}fracOut)\end{aligned}$$

and $r = A + P \Leftrightarrow gOvfl = sOvfl = 0$, which is the condition under which $R[IXC]$ is not set.

Subcase 3.2: $ovfl2 = 0$ and $ovfl = 1$.

If $expo(sumShft) = 107$, then by Lemma 17.23, $expo(sumOvfl) = 52$ and

$$sumOvfl = 2^{52} + sumOvfl[51:0] = 2^{52} + fracOut.$$

Since $expShft > 0$, $expOut = expShft + 1$, and by Lemma 17.21,

$$\begin{aligned}|r| &= 2^{expShft-(2^{10}-1)-51}sumOvfl \\ &= 2^{expShft+1-(2^{10}-1)}(2^{-52}sumOvfl) \\ &= 2^{expOut-(2^{10}-1)}(1+2^{-52}fracOut)\end{aligned}$$

and $r = A + P \Leftrightarrow gOvfl = sOvfl = 0$, which is the condition under which $R[IXC]$ is not set.

On the other hand, if $expo(sumShft) = 106$, then by Lemma 17.24, $sumNorm = 2^{53}$, $gOvfl = 1$, and either $gNorm = 1$ or $sNorm = 1$. If $expShft > 0$, then $expOut = expShft + 1$, and by Lemma 17.21,

$$\begin{aligned}|r| &= 2^{expShft-(2^{10}-1)-52}sumNorm \\ &= 2^{expShft+1-(2^{10}-1)} \\ &= 2^{expOut-(2^{10}-1)}(1+2^{-52}fracOut).\end{aligned}$$

Similarly, if $expShft = 0$, then by Lemma 17.22,

$$\begin{aligned}|r| &= 2^{1-(2^{10}-1)-52}sumNorm \\ &= 2^{expShft-(2^{10}-1)-52}2^{53} \\ &= 2^{2-(2^{10}-1)} \\ &= 2^{expOut-(2^{10}-1)}(1+2^{-52}fracOut).\end{aligned}$$

In either case, since either $gNorm = 1$ or $sNorm = 1$, $r \neq A + P$, and since $gOvfl = 1$, $R[IXC]$ is set.
Subcase 3.3: $ovfl2 = ovfl = 0$.

By Lemmas 17.23 and 17.24, $expo(sumShft) = 106$ and $expo(sumNorm) = 52$. Since $fracOut = sumNorm[51:0]$,

$$sumNorm = 2^{52} + sumNorm[51:0] = 2^{52}(1 + 2^{-52}fracOut).$$

If $expo(expShft) > 0$, then $expOut = expShft$, and by Lemma 17.21,

$$|r| = 2^{expShft-(2^{10}-1)-52}sumNorm = 2^{expOut-(2^{10}-1)}(1 + 2^{-52}fracOut).$$

Similarly, if $expo(expShft) = 0$, then $expOut = 1$, and by Lemma 17.22,

$$|r| = 2^{1-(2^{10}-1)-52}sumNorm = 2^{expOut-(2^{10}-1)}(1 + 2^{-52}fracOut).$$

In either case, $r = A + P \Leftrightarrow gNorm = sNorm = 0$, which is the condition under which $R[IXC]$ is not set. □

We have the following correctness theorem for FADD. For the case of numerical operands with a nonzero sum, the behavior of the data result and the post-computation exception flags follow from Lemma 17.26. The behavior of the pre-computation flags and the remaining special cases are readily verified by a straightforward case analysis comparing *fadd64* with the specification function *arm-binary-spec*:

Theorem 17.1 *Let*

$$\langle D_{spec}, R_{spec}\rangle = arm\text{-}binary\text{-}spec(ADD, opa, opp[116:53], R_{in}, DP),$$

where opa is a 64-bit vector, opp is a 117-bit vector with $opp[52:0] = 0$*, and* R_{in} *is a 32-bit vector. Let*

$$\langle D, R\rangle = fadd64(opa, opp, fz, dn, rmode, fma, inz, piz, mulOvfl, mulExcps),$$

where $fz = R_{in}[24]$, $dn = R_{in}[25]$, $rmode = R_{in}[23:22]$, $fma = 0$*, and inz, piz, mulOvfl, and mulExcps are arbitrary.*

Then $D = D_{spec}$ *and* $R_{in} \mid flags = R_{spec}$.

According to the following theorem, FMA is correctly implemented by a combination of *fadd64* and *fmul64*. This similarly follows from Lemmas 17.26, 16.11, and 16.12 and a comparison of these functions with *arm-fma-spec*:

Theorem 17.2 *Let*

$$\langle D_{spec}, R_{spec}\rangle = arm\text{-}fma\text{-}spec(opa, opb, opc, R_{in}, DP),$$

where opa, opb, and opc are 64-bit vectors and R_{in} is a 32-bit vector. Let

$$\langle opp, mulExcps, piz, inz, mulOvfl\rangle = fmul64(opb, opc, fz, dn, rmode, fma),$$

where $fz = R_{in}[24]$, $dn = R_{in}[25]$, $rmode = R_{in}[23:22]$, *and* $fma = 1$. *Let*

$$\langle D, flags\rangle = fadd64(opa, opp, fz, dn, rmode, fma, inz, piz, mulOvfl, mulExcps).$$

Then $D = D_{spec}$ *and* $R_{in} \mid flags = R_{spec}$.

Chapter 18
Multi-Precision Radix-4 SRT Division

Unlike the dedicated double-precision multiplier and adder described in the preceding two chapters, a single module of our FPU performs floating-point division and square root extraction at all three precisions: double, single, and half. This module is modeled, however, by two separate functions, `fdiv64` and `fsqrt64`, the first of which, displayed in Appendix D, is the subject of this chapter. This function is based on the implementation of the minimally redundant radix-4 case of SRT division that is addressed by Lemma 10.7 of Sect. 10.2.

The iterative phase of the algorithm is naturally modeled in C as a `for` loop. As usual, successful equivalence checking between the model and the RTL requires faithful modeling of the essential computations, which in this case means that the partial remainder and quotient must be replicated precisely at each iteration. With the goal of minimizing latency, the RTL executes three iterations on each cycle, and in order to achieve this, different approximations are used for the iterations within a cycle. Consequently, each iteration of the `for` loop corresponds to a cycle, i.e., three iterations of the algorithm rather than one.

18.1 Overview

The input parameters of the top-level function `fdiv64` are as follows:

- `ui64 opa, opb`: Encodings of the dividend and divisor, respectively. For formats *SP* and *HP*, the operands reside in the low-order bits.
- `ui2 fmt`: A 2-bit encoding of a floating-point format ($DP = 2$, $SP = 1$, $HP = 0$).
- `bool fz, dn`: The FZ and DN fields of the FPSCR (Sect. 14.1).
- `ui2 rmode`: The RC field of the FPSCR, a 2-bit encoding of an IEEE rounding mode (Table 14.1), which we shall denote as $\mathcal{R}$.

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_18

Two results are returned:

- `ui64 D`: The data result. For $fmt = SP$ (resp., HP), $D[31:0]$ (resp., $D[15:0]$) holds the data, and the higher bits are 0.
- `ui8 flags`: The exception flags, $FPSCR[7:0]$. We use the mnemonics defined in Fig. 14.1 to refer to the bits of this vector.

We also define the following values:

- The operand values, $A = decode(opa, fmt)$ and $B = decode(opb, fmt)$;
- The precision of the operation, $p = \begin{cases} 11 \text{ if } fmt = HP \\ 24 \text{ if } fmt = SP \\ 53 \text{ if } fmt = DP; \end{cases}$
- The exponent width, $e = \begin{cases} 5 \text{ if } fmt = HP \\ 8 \text{ if } fmt = SP \\ 11 \text{ if } fmt = DP; \end{cases}$
- The exponent $bias = 2^{e-1} - 1$;

The initial phase of `fdiv64` handles the early termination cases, in which at least one operand is a zero, a NaN, or an infinity. Since these cases are trivial, we shall focus on the remaining computational case, in which each operand is either a denormal that is not forced to 0 or normal.

Before entering the `for` loop, the operands are normalized if necessary and prescaled according to (10.7), and the first iteration of the algorithm of Sect. 10.1 is executed. Since each of the N iterations of the loop corresponds to an RTL cycle in which three iterations of the algorithm are executed, the number of iterations of the algorithm is $3N + 1$.

Notation For each variable that is updated within the `for` loop, we shall use the subscript j to denote the value produced by the jth iteration of the algorithm.

In particular, the value of the jth quotient digit is q_j. Once we define the prescaled values x and d of A and B, the partial quotients Q_j and remainders R_j will also be defined, according to (10.2) and (10.3). The partial remainder is represented in redundant signed-digit form by RP and RN, 59-bit vectors with 3 implicit integer bits. That is, we shall show that

$$2^{56} R_j \equiv RP_j - RN_j \pmod{2^{59}}. \tag{18.1}$$

To see that 3 integer bits are sufficient to represent the full range of remainders, note that the bounds on d (10.7) together with Lemma 10.7(a) yield

$$|R_j| \leq \frac{2}{3} d \leq \frac{2}{3} \cdot \frac{9}{8} = \frac{3}{4} \tag{18.2}$$

and hence $|4R_j| \leq 3$.

The partial quotient is also represented in redundant signed-digit form, by the vectors *QP* and *QN*, each consisting of 54 fractional bits. The integer bits of the quotient are not explicitly stored. Thus, we shall show that

$$4^{j-1}(Q_j - q_1) = QP_j - QN_j. \tag{18.3}$$

As specified in Lemma 10.7, the derivation of the quotient digit q_{j+1} is based on an approximation A_j of the remainder R_j, which is compared with the constants of Fig. 10.1. As is typical of SRT designs, when the $(j+1)$st iteration is either the first or the second iteration of a cycle, A_j is derived from the leading bits of RP_j and RN_j, in this case by means of a 6-bit adder. In order to satisfy timing constraints, however, when the $(j+1)$st iteration is the third iteration of a cycle, A_j is computed in two steps directly from RP_{j-1}, RN_{j-1}, and q_j, before RP_j and RN_j are available. The analysis of this critical feature of the design appears in the proof of Lemma 18.5.

18.2 Pre-processing

The function *normalize* performs a mantissa shift for each denormal operand, using the same auxiliary function *CLZ53* as the multiplier of Chap. 16. The values computed by normalize, representing the operand significands and the predicted exponent of the result, satisfy the following:

Lemma 18.1

(a) $siga = 2^{52}sig(A)$;
(b) $sigb = 2^{52}sig(B)$;
(c) $\left|\frac{A}{B}\right| = 2^{expDiff-bias}\left(\frac{siga}{sigb}\right)$.

Proof It is clear that $2^{52} \leq siga < 2^{53}$ and $|A| = 2^{expaShft-bias-52}siga$: if *opa* is normal this is trivial, and the denormal case follows from Lemma 16.6. It follows that $siga = 2^{52}sig(A)$. The analogous results hold for B. Thus,

$$\left|\frac{A}{B}\right| = 2^{expaShft-expbShft}\left(\frac{siga}{sigb}\right),$$

where

$$expaShft - expbShft = (expaShft - expbShft + bias) - bias = expDiff - bias. \square$$

Before entering the iterative phase, the operands are prescaled by a multiplier M based on the three most significant fractional bits of $sig(B)$, i.e.,

$$sigb[51:49] = \lfloor 8(sig(B) - 1\rfloor.$$

$sigb[51{:}49]$	M
000	$2 = \frac{1}{2} + \frac{1}{2} + 1$
001	$\frac{7}{4} = \frac{1}{4} + \frac{1}{2} + 1$
010	$\frac{13}{8} = \frac{1}{2} + \frac{1}{8} + 1$
011	$\frac{3}{2} = \frac{1}{2} + 0 + 1$
100	$\frac{11}{8} = \frac{1}{4} + \frac{1}{8} + 1$
101	$\frac{5}{4} = \frac{1}{4} + 0 + 1$
110	$\frac{9}{8} = 0 + \frac{1}{8} + 1$
111	$\frac{9}{8} = 0 + \frac{1}{8} + 1$

Fig. 18.1 Prescaling multiplier M

The values of M corresponding to the eight possible values of this index are displayed in Fig. 18.1 as a sum of powers of 2. These terms will correspond to the partial products in the multiplication of M by both *sigb* and *siga*.

We define

$$d = M\frac{sig(B)}{2} \tag{18.4}$$

and

$$x = \begin{cases} M\frac{sig(A)}{2} & \text{if } sig(\text{A}) \geq sig(\text{B}) \\ Msig(A) & \text{if } sig(\text{A}) < sig(\text{B}). \end{cases} \tag{18.5}$$

Q_j and R_j are now defined by (10.2) and (10.3).

We verify the required bounds on d and x:

Lemma 18.2 $\frac{63}{64} \leq d \leq \frac{9}{8}$ *and* $d \leq x < 2d$.

Proof The bounds on d may be established for each value of $sigb[51:49] = \lfloor 8(sig(B) - 1) \rfloor$ separately. For example, if $sigb[51:49] = 3$, then $3 \leq 8(sig(B) - 1) < 4$, or

$$\frac{11}{8} \leq sig(B) < \frac{3}{2}.$$

In this case, $M = \frac{3}{2}$, and according to (18.4), $d = \frac{3}{4}sig(B)$ and

$$\frac{63}{64} < \frac{33}{32} = \frac{3}{4} \cdot \frac{11}{8} \leq d \leq \frac{3}{4} \cdot \frac{3}{2} = \frac{9}{8}.$$

The other seven cases are similar.

To derive the bounds on x, note that if $sig(A) \geq sig(B)$, then $x = d \cdot sig(A)/sig(B)$, where $1 \leq sig(A)/sig(B) < 2$, and the claim follows. Similarly, if $sig(A) < sig(B)$, then $x = 2d \cdot sig(A)/sig(B)$, where $\frac{1}{2} \leq sig(A)/sig(B) < 1$. □

In addition to the prescaling, the function *prescale* performs the first iteration of the algorithm, returning a non-redundant representation of d, a signed-digit redundant representation of the partial remainder R_1, the digit q_1, and the biased exponent *expQ* of the unrounded quotient. Note that $QP_1 = QN_1 = 0$, which is consistent with Eqs. (18.3) and (10.2).

Lemma 18.3

(a) $2^{56}d = div$;
(b) q_1 *is the greatest* $k \in \{-2, \ldots, 2\}$ *such that* $m_k \leq A_0$*, for some* $A_0 \in \mathbb{Q}$ *that satisfies* $8A \in \mathbb{Z}$ *and* $|A_0 - 4R_0| < \frac{1}{8}$.
(c) $|R_1| \leq \frac{2}{3}d$;
(d) $2^{56}R_1 = RP_1 - RN_1$ *and* $RP_1[52 - p : 0] = RN_1[52 - p : 0] = 0$;
(e) $\left|\frac{A}{B}\right| = 2^{expQ-bias}\left(\frac{x}{d}\right)$.

Proof

(a) The table of Fig. 18.1 expresses the prescaling multiplier as a sum $M = t_1 + t_2 + t_3$, where $t_1 \in \{0, \frac{1}{4}, \frac{1}{2}\}$, $t_2 \in \{0, \frac{1}{8}, \frac{1}{2}\}$, and $t_3 = 1$. Referring to the definition of *prescale*, we see that for each value of $sigb[51 : 49]$, and for $i \in \{1, 2, 3\}$, $divi = 8t_i sigb$. Thus,

$$div1 + div2 + div3 = 8(t_1 + t_2 + t_3)sigb = 8Msigb = 2^{55}Msig(B) = 2^{56}d.$$

Since $div1[55] = div2[55] = 0$, only a 0 bit is shifted out in the construction of *divCar* and the sum is preserved by the 3:2 compressor. Thus,

$$div = divSum + divCar = 2^{56}d.$$

(b) Let $A_0 = \frac{1}{8}remBits$. First, we must show that $|A_0 - 4R_0| < \frac{1}{8}$, i.e., $|remBits - 32R_0| = |remBits - 8x| < 1$. By the same argument as used above,

$$remSum + remCar = 2^{55}Msig(A) = \begin{cases} 2^{56}x \text{ if } sig(\text{A}) \geq sig(\text{B}) \\ 2^{55}x \text{ if } sig(\text{A}) < sig(\text{B}). \end{cases}$$

Suppose $sig(A) < sig(B)$. Then $2^{52}remBits = 2^{52}remSum[55:52] + 2^{52}remCar[55:52]$ and by case analysis on $remSum[51]$ and $remCar[51]$,

$$|2^{52}remBits - (remSum + remCar)| = |2^{52}remBits - 2^{55}x| < 2^{52},$$

or $|remBits - 8x| < 1$.

Similarly, in the case $sig(A) \geq sig(B)$, we have $|2^{53}remBits - 2^{56}x| < 2^{53}$ and reach the same conclusion.

To prove the second claim, note that according to the definition of *prescale*, since $8m_2 = 13$,

$$q_1 = \begin{cases} 2 \text{ if } 8m_2 \leq remBits \\ 1 \text{ if } 8m_2 > remBits. \end{cases}$$

Therefore, it suffices to show that $8m_1 \leq remBits$. But

$$remBits > 8x - 1 \geq 8d - 1 \geq 8 \cdot \frac{63}{64} - 1 > 4 = 8m_1.$$

(c) This follows from (b), (10.2), and Lemma 10.7 with $j = 0$.
(d) $RP_1 - RN_1 = 2^{56}x - q_1 div = 2^{56}(4R_0 - q_1 d) = 2^{56}R_1$, and it follows from $siga[52-p:0] = sigb[52-p:0] = 0$ that $RP_1[52-p:0] = RN_1[52-p:0] = 0$.
(e) Since Lemma 18.1 (c) holds prior to execution of *prescale*, during which *expDiff* is decremented just in case $siga < sigb$, we need only observe that, according to (18.4) and (18.5),

$$\frac{x}{d} = \begin{cases} \frac{siga}{sigb} & \text{if } siga \geq sigb \\ \frac{2siga}{sigb} & \text{if } siga < sigb. \end{cases} \square$$

Lemma 18.4 *For* $1 \leq j \leq 3N+1$, $2^{56}R_j \in \mathbb{Z}$.

Proof This follows from Eqs. (10.2) and (10.3) and Lemma 18.3(a). □

18.3 Iterative Phase

Lemma 18.5 *The following conditions hold for all* j, $1 \leq j \leq 3N+1$*:*

(a) q_j *is the greatest* $k \in \{-2, \ldots, 2\}$ *such that* $m_k \leq A_{j-1}$*, for some* $A_{j-1} \in \mathbb{Q}$ *that satisfies* $8A_{j-1} \in \mathbb{Z}$ *and* $|A_{j-1} - 4R_{j-1}| < \frac{1}{8}$.
(b) $|R_j| \leq \frac{2}{3}d$;
(c) $2^{56}R_j \equiv RP_j - RN_j \pmod{2^{59}}$ *and* $RP_j[52-p:0] = RN_j[52-p:0] = 0$;
(d) $4^{j-1}(Q_j - q_1) = QP_j - QN_j$.

Proof The proof proceeds by induction on j. For the case $j = 1$, (a), (b), and (c) are stated in Lemma 18.3, and (d) holds trivially, since $Q_1 = q_1$ and $QP_1 = QN_1 = 0$. Let $1 \leq j \leq 3N$. We assume the lemma holds for all ℓ such that $1 \leq \ell \leq j$ and show that it holds for $j + 1$.

(a) The quotient digit q_{j+1} is computed as $nextDigit(remS6_{j+1})$, where $remS6_{j+1}$ is a 6-bit vector computed by one of the functions *iter1*, *iter2*, and *iter3*. It is readily verified that the digit returned by *nextDigit* is the greatest $k \in \{-2, \ldots, 2\}$ such that $8m_k \leq si(remS6_{j+1}, 6)$. Let $A_j = \frac{1}{8}si(remS6_{j+1}, 6)$. We need only show that $|A_j - 4R_j| < \frac{1}{8}$.

When iteration $j + 1$ occurs as either the first or second iteration of a cycle, *nextDigit* is called with

$$remS6_{j+1} = RP_j[56:51] - RN_j[56:51] \mod 2^6.$$

In this case, we invoke Lemma 2.54 with

$$X = RP_j[56:0] - RN_j[56:0] \equiv 2^{56}R_j \pmod{2^{57}},$$

$$Y = 2^{51}(RP_j[56:51] - RN_j[56:51]) \equiv 2^{51}remS6_{j+1} \pmod{2^{57}},$$

and $n = 57$. Since $|X - Y| = |RP_j[50:0] - RN_j[50:0]| < 2^{51}$ and

$$|2^{56}R_j| \leq 2^{54} \cdot 3 = 2^{56} - 2^{54} < 2^{56} - |X - Y|,$$

$$si(X \mod 2^{57}, 57) = 2^{56}R_j,$$

$$\begin{aligned}|si(X \mod 2^{57}, 57) - si(Y \mod 2^{57}, 57)| &= |2^{56}R_j - si(2^{51}remS6_{j+1}, 57)| \\ &= |X - Y| \\ &< 2^{51},\end{aligned}$$

and division by 2^{51} yields

$$|4R_j - A_j| = \left|4R_j - \frac{1}{8}si(remS6_{j+1}, 6)\right| < \frac{1}{8}.$$

When iteration $j + 1$ is the third iteration of a cycle, the construction of $remS6_{j+1}$ involves two successive approximations, computed as by-products of the two preceding iterations. The first iteration of the cycle computes, along with a redundant representation of R_{j-1}, the 9-bit sum

$$RS9_{j-1} = RP_{j-1}[56:48] - RN_{j-1}[56:48] \mod 2^9,$$

representing an approximation of $4R_{j-1}$. Let

$$R = 4RP_{j-1} - 4RN_{j-1}$$

and

$$\tilde{R} = 2^{50}(RP_{j-1}[58:48] - RN_{j-1}[58:48]).$$

Then

$$R \equiv 2^{58}R_{j-1} \pmod{2^{59}},$$

$$RS9_{j-1} \equiv 2^{-50}\tilde{R} \pmod{2^9},$$

and

$$|R - \tilde{R}| = |4(RP_{j-1}[47:0] - RN_{j-1}[47:0])| < 2^{50}.$$

The second iteration computes (in parallel with a redundant representation of R_j) the 7-bit sum $RS7_j$, derived from $RS9_{j-1}$ and q_j and representing an approximation of R_j. In the third iteration, *nextDigit* is called with $remS6_{j+1} = RS7_j[6:1]$.

Let

$$D = -q_j div$$

and

$$\tilde{D} = \begin{cases} -q_j div & \text{if } q_j \leq 0 \\ -q_j div - 1 & \text{if } q_j > 0 \end{cases}$$

Clearly, $0 \leq D - \tilde{D} \leq 1$. Let $S = \lfloor 2^{-51}\tilde{R} \rfloor + \lfloor 2^{-51}\tilde{D} \rfloor + c$, where

$$c = \begin{cases} 0 \text{ if } \tilde{R}[50] = \tilde{D}[50] = 0 \\ 1 \text{ otherwise.} \end{cases}$$

Now

$$\begin{aligned} RS7_j &= (RS9_{j-1} + \lfloor 2^{-50}\tilde{D} \rfloor + 1) \bmod 2^7 \\ &= (2^{-50}\tilde{R} + \lfloor 2^{-50}\tilde{D} \rfloor + 1) \bmod 2^7 \\ &= \left((2\lfloor 2^{-51}\tilde{R} \rfloor + \tilde{R}[50]) + (2\lfloor 2^{-51}\tilde{D} \rfloor + \tilde{D}[50]) + 1\right) \bmod 2^7 \end{aligned}$$

and therefore, by Lemma 1.21,

$$\begin{aligned}
remS6_{j+1} &= RS7_j[6:1] \\
&= \left\lfloor \frac{1}{2} RS7_j \right\rfloor \\
&= \left(\lfloor 2^{-51}\tilde{R} \rfloor + \lfloor 2^{-51}\tilde{D} \rfloor + \left\lfloor \frac{1}{2}(\tilde{R}[50] + \tilde{D}[50] + 1) \right\rfloor \right) \bmod 2^6 \\
&= S \bmod 2^6.
\end{aligned}$$

We shall invoke Lemma 2.54 with $X = R + D$, $Y = 2^{51}S$, and $n = 57$. Thus,

$$X \equiv 2^{58}R_{j-1} - q_j div = 2^{56}(4R_{j-1} - q_j d) = 2^{56}R_j \pmod{2^{59}}$$

so that once again, $si(X \bmod 2^{57}, 57) = 2^{56}R_j$, and

$$\begin{aligned}
si(Y \bmod 2^{57}, 57) &= si(2^{51}S \bmod 2^{57}, 57) \\
&= 2^{51} si(S \bmod 2^6, 6) \\
&= 2^{51} si(remS6_{j+1}, 6).
\end{aligned}$$

Since

$$\tilde{R} = 2^{51}\lfloor 2^{-51}\tilde{R} \rfloor + \tilde{R}[50:0] = 2^{51}\lfloor 2^{-51}\tilde{R} \rfloor + 2^{50}\tilde{R}[50]$$

and

$$\tilde{D} = 2^{51}\lfloor 2^{-51}\tilde{D} \rfloor + \tilde{D}[50:0] = 2^{51}\lfloor 2^{-51}\tilde{F} \rfloor + 2^{50}\tilde{D}[50] + \tilde{D}[49:0],$$

$$\tilde{R} + \tilde{D} - 2^{51}S = 2^{50}(\tilde{R}[50] + \tilde{D}[50] - 2c) + \tilde{D}[49:0].$$

By considering all possible combinations of $\tilde{R}[50]$ and $\tilde{D}[50]$ and the range of values $0 \le \tilde{D}[49:0] < 2^{50}$, it follows that $|\tilde{R} + \tilde{D} - 2^{51}S| < 2^{50}$. Thus,

$$\begin{aligned}
|X - Y| &= |R + D - 2^{51}S| \\
&\le |\tilde{R} + \tilde{D} - 2^{51}S| + |R - \tilde{R}| + |D - \tilde{D})| \\
&< 2^{50} + (2^{50} - 1) + 1 \\
&= 2^{51},
\end{aligned}$$

which implies

$$|si(X \bmod 2^{57}, 57) - si(Y \bmod 2^{57}, 57)| = |2^{56}R_j - 2^{51}si(remS6_{j+1}, 6)| < 2^{51},$$

and

$$|4R_j - A_j| = \left|4R_j - \frac{1}{8}si(remS6_{j+1}, 6)\right| < \frac{1}{8}.$$

(b) This follows from (a) and Lemma 10.7.

(c) By induction and the definition of *nextRem*,

$$RP_{j+1}[52 - p : 0] = RN_{j+1}[52 - p : 0] = 0;$$

we must show that $2^{56}R_{j+1} \equiv RP_{j+1} - RN_{j+1} \pmod{2^{59}}$.

We refer to the local variables of *nextRem*. Let

$$\epsilon = \begin{cases} 0 \text{ if } q_{j+1} \leq 0 \\ 1 \text{ if } q_{j+1} > 0. \end{cases}$$

Since

$$div[58 : 53 - p] = div[56 : 53 - p] = 2^{p+3}d,$$

it is clear that for all values of q_{j+1},

$$\begin{aligned} divMult_{j+1}[58 : 53 - p] &\equiv -q_{j+1}div[58 : 53 - p] - \epsilon \\ &= -2^{p+3}q_{j+1}d - \epsilon \pmod{2^{p+6}}. \end{aligned}$$

ss Since $RP_j[52 - p : 0] = RN_j[52 - p : 0] = 0$, Lemma 1.18 implies

$$\begin{aligned} &2^{53-p}\left((RP4_{j+1}[58 : 53 - p] - RN4_{j+1}[58 : 53 - p]) \bmod 2^{p+6}\right) \\ &= \left(2^{53-p}(RP4_{j+1}[58 : 53 - p] - RN4_{j+1}[58 : 53 - p])\right) \bmod 2^{59} \\ &= (RP4_{j+1} - RN4_{j+1}) \bmod 2^{59} \\ &= \left(4(RP_j - RN_j)\right) \bmod 2^{59} \\ &= (2^{58}R_j) \bmod 2^{59} \\ &= 2^{53-p}\left((2^{p+5}R_j) \bmod 2^{p+6}\right), \end{aligned}$$

or

$$RP4_{j+1}[58 : 53 - p] - RN4_{j+1}[58 : 53 - p] \equiv 2^{p+5}R_j \pmod{2^{p+6}}.$$

By Lemma 8.4,

$$\begin{aligned}
&\tilde{}sum_{j+1}[58:53\text{–}p] + 2car_{j+1}[58:53\text{–}p] \\
&= RP4_{j+1}[58:53\text{–}p] + \sim RN4_{j+1}[58:53\text{–}p] + divMult_{j+1}[58:53\text{–}p] \\
&\equiv RP4_{j+1}[58:53\text{–}p] - RN4_{j+1}[58:53\text{–}p] - 1 + divMult_{j+1}[58:53\text{–}p] \\
&\equiv 2^{p+5}R_j - 1 - 2^{p+3}q_{j+1}d - \epsilon \\
&\equiv 2^{p+3}(4R_j - q_{j+1}d) - 1 - \epsilon \\
&\equiv 2^{p+3}R_{j+1} - 1 - \epsilon \pmod{2^{p+6}}.
\end{aligned}$$

Now

$$\begin{aligned}
&(RP_{j+1} - RN_{j+1}) \bmod 2^{59} \\
&= \left(2^{53-p}(RP_{j+1}[58:53-p] - RN_{j+1}[58:53-p])\right) \bmod 2^{59} \\
&= 2^{53-p}\left((RP_{j+1}[58:53-p] - RN_{j+1}[58:53-p]) \bmod 2^{p+6}\right),
\end{aligned}$$

where

$$\begin{aligned}
RP_{j+1}[58:53-p] &= 2RP_{j+1}[58:54-p] + RP_{j+1}[53-p] \\
&= 2car_{j+1}[57:53-p] + \epsilon \\
&= 2(car_{j+1}[58:53-p] \bmod 2^{p+5}) + \epsilon \\
&= (2car_{j+1}[58:53-p]) \bmod 2^{p+6}) + \epsilon
\end{aligned}$$

and

$$\begin{aligned}
-RN_{j+1}[58:53-p] &= -sum_{j+1}[58:53-p] \\
&\equiv \tilde{}sum_{j+1}[58:53-p] + 1 \pmod{2^{p+6}}.
\end{aligned}$$

Thus,

$$\begin{aligned}
&(RP_{j+1} - RN_{j+1}) \bmod 2^{59} \\
&= 2^{53-p}\left((2car_{j+1}[58:53-p]+\epsilon+\tilde{}sum_{j+1}[58:53-p]+1) \bmod 2^{p+6}\right) \\
&= 2^{53-p}\left((2^{p+3}R_{j+1} - 1 - \epsilon + \epsilon + 1) \bmod 2^{p+6}\right) \\
&= 2^{53-p}\left((2^{p+3}R_{j+1}) \bmod 2^{p+6}\right) \\
&= (2^{56}R_{j+1}) \bmod 2^{59}.
\end{aligned}$$

(d) This is a simple consequence of (10.2), the definition of *nextQuot*, and induction. □

We have the following error bound for the final quotient:

Lemma 18.6 $\left|Q - \frac{x}{d}\right| \leq \frac{2}{3} \cdot 2^{-6N}$.

Proof This is an immediate consequence of Lemma 18.5 (b) with $j = 3N + 1$ and (10.3). □

18.4 Post-Processing and Rounding

Notation In discussing the final loop variable values RP_{3N+1}, RN_{3N+1}, QP_{3N+1}, and QN_{3N+1}, the subscript may be omitted. We shall similarly abbreviate the final quotient Q_{3N+1} and remainder R_{3N+1} as Q and R.

The rounder selects one of two values returned by *computeQ*: the truncated quotient *Qtrunc* or the incremented truncated quotient *Qinc*. The third value of *computeQ* is an indication of inexactness, *stk*.

In the trivial case of division by a power of 2, the quotient is exact and *Qtrunc* is selected:

Lemma 18.7 *Assume* $divPow2 = 1$.

(a) $2^p \cdot \frac{x}{d} \in Z$;
(b) $Qtrunc \equiv 2^p \cdot \frac{x}{d} \pmod{2^p}$;
(c) $stk = 0$.

Proof In this case, $sig(B) = 1$ and by Definitions (18.4) and (18.5),

$$2^p \cdot \frac{x}{d} = 2^p sig(A) \in Z.$$

By Lemma 5.1 (c),

$$sig(A) = 1 + 2^{1-p} mana,$$

and hence,

$$2^p \cdot \frac{x}{d} = 2^p sig(A) = 2^p + 2mana = 2^p + Qtrunc \equiv Qtrunc \pmod{2^p}.$$

By definition, $stk = 0$. □

We turn to the usual case $divPow2 = 0$:

Lemma 18.8 *Assume* $divPow2 = 0$.

(a) $Qtrunc \equiv \left\lfloor 2^p \cdot \frac{x}{d} \right\rfloor \pmod{2^p}$;
(b) $Qinc \equiv \left\lfloor 2^p \cdot \frac{x}{d} \right\rfloor + 2 \pmod{2^p}$.

Proof We consider the execution of *computeQ*(*QP*, *QN*, *RP*, *RN*, *fmt*, *false*).

It is clear that

$$Q0 \equiv QP - QN - 1 \pmod{2^{54}} \tag{18.6}$$

and

$$Q1inc \equiv QP - QN + inc \pmod{2^{54}}. \tag{18.7}$$

We would like to show that

$$Q1 \equiv QP - QN \pmod{2^{54}} \tag{18.8}$$

and

$$Q0inc \equiv QP - QN - 1 + inc \pmod{2^{54}}. \tag{18.9}$$

But it is clear that the same congruences do hold modulo 8, and therefore we need only show that

$$\left\lfloor \frac{Q1}{8} \right\rfloor \equiv \left\lfloor \frac{QP - QN}{8} \right\rfloor \pmod{2^{51}} \tag{18.10}$$

and

$$\left\lfloor \frac{Q0inc}{8} \right\rfloor \equiv \left\lfloor \frac{QP - QN - 1 + inc}{8} \right\rfloor \pmod{2^{51}}, \tag{18.11}$$

for then

$$\begin{aligned} Q1 &= 8\left\lfloor \frac{Q1}{8} \right\rfloor + Q1 \bmod 8 \\ &\equiv 8\left\lfloor \frac{QP - QN}{8} \right\rfloor + (QP - QN) \bmod 8 \\ &= QP - QN \pmod{2^{54}}, \end{aligned}$$

and an analogous argument yields (18.9). We shall appeal to the simple observation that for integers z, k, ℓ, and n, if $0 < k \leq \ell \leq n$, then

$$\left\lfloor \frac{z+k}{n} \right\rfloor = \begin{cases} \lfloor \frac{z}{n} \rfloor & \text{if } z + k \bmod n \geq k \\ \lfloor \frac{z+\ell}{n} \rfloor & \text{if } z + k \bmod n < k. \end{cases} \tag{18.12}$$

For the proof of (18.10), we instantiate (18.12) with $z = QP - QN - 1$, $k = 1$, $\ell = 1 + inc$, and $n = 8$. If $QP - QN \bmod 8 = 0$, i.e, $z + k \bmod n < k$, then

$$\left\lfloor \frac{QP - QN}{8} \right\rfloor = \left\lfloor \frac{z + k}{n} \right\rfloor = \left\lfloor \frac{z + \ell}{n} \right\rfloor = \left\lfloor \frac{QP - QN + inc}{8} \right\rfloor$$

and according to the definition of $Q1$,

$$\begin{aligned}
\left\lfloor \frac{Q1}{8} \right\rfloor \bmod 2^{51} &= Q1[53:3] \\
&= Q1inc[53:3] \\
&= (QP - QN + inc)[53:3] \\
&= \left\lfloor \frac{QP - QN + inc}{8} \right\rfloor \bmod 2^{51} \\
&= \left\lfloor \frac{QP - QN}{8} \right\rfloor \bmod 2^{51}.
\end{aligned}$$

In the remaining case,

$$\left\lfloor \frac{QP - QN}{8} \right\rfloor = \left\lfloor \frac{z + k}{n} \right\rfloor = \left\lfloor \frac{z}{n} \right\rfloor = \left\lfloor \frac{QP - QN - 1}{8} \right\rfloor$$

and

$$\begin{aligned}
\left\lfloor \frac{Q1}{8} \right\rfloor \bmod 2^{51} &= \left\lfloor \frac{Q0}{8} \right\rfloor \bmod 2^{51} \\
&= \left\lfloor \frac{QP - QN - 1}{8} \right\rfloor \bmod 2^{51} \\
&= \left\lfloor \frac{QP - QN}{8} \right\rfloor \bmod 2^{51}.
\end{aligned}$$

For the proof of (18.11), we instantiate (18.12) with $z = QP - QN - 1$, $k = inc$, $\ell = 1 + inc$, and $n = 8$. If $QP - QN + inc - 1 \bmod 8 < inc$, i.e, $z + k \bmod n < k$, then

$$\left\lfloor \frac{QP - QN + inc - 1}{8} \right\rfloor = \left\lfloor \frac{z + k}{n} \right\rfloor = \left\lfloor \frac{z + \ell}{n} \right\rfloor = \left\lfloor \frac{QP - QN + inc}{8} \right\rfloor$$

and according to the definitions of $Q0inc$ and inc,

$$\left\lfloor \frac{Q0inc}{8} \right\rfloor \bmod 2^{51} = Q0inc[53:3]$$

$$
\begin{aligned}
&= \mathit{Q1inc}[53:3] \\
&= (\mathit{QP} - \mathit{QN} + \mathit{inc})[53:3] \\
&= \left\lfloor \frac{\mathit{QP} - \mathit{QN} + \mathit{inc}}{8} \right\rfloor \bmod 2^{51} \\
&= \left\lfloor \frac{\mathit{QP} - \mathit{QN} + \mathit{inc} - 1}{8} \right\rfloor \bmod 2^{51}.
\end{aligned}
$$

Otherwise,

$$
\left\lfloor \frac{\mathit{QP} - \mathit{QN} + \mathit{inc} - 1}{8} \right\rfloor = \left\lfloor \frac{z+k}{n} \right\rfloor = \left\lfloor \frac{z}{n} \right\rfloor = \left\lfloor \frac{\mathit{QP} - \mathit{QN} - 1}{8} \right\rfloor
$$

and

$$
\begin{aligned}
\left\lfloor \frac{\mathit{Q0inc}}{8} \right\rfloor \bmod 2^{51} &= \left\lfloor \frac{\mathit{Q0}}{8} \right\rfloor \bmod 2^{51} \\
&= \left\lfloor \frac{\mathit{QP} - \mathit{QN} - 1}{8} \right\rfloor \bmod 2^{51} \\
&= \left\lfloor \frac{\mathit{QP} - \mathit{QN} + \mathit{inc} - 1}{8} \right\rfloor \bmod 2^{51}.
\end{aligned}
$$

Next, the sign of the final remainder is used to select either $\mathit{Q1}$ and $\mathit{Q1inc}$ or $\mathit{Q0}$ and $\mathit{Q0inc}$. By Lemma 18.5 (c), $\mathit{rem} = 2^{56}R \bmod 2^{59}$. It follows from Lemma 18.5 (b) that $\mathit{remZero} = 1 \Leftrightarrow R = 0$ and $2^{56}R = \mathit{si}(\mathit{rem}, 59)$, which implies $\mathit{remSign} = 0 \Leftrightarrow R \geq 0$.

By Lemma 18.6, $|2^{6N}Q - 2^{6N}\frac{x}{d}| < 1$. If $R \geq 0$, then

$$
2^{6N}Q \leq 2^{6N}\frac{x}{d} < 2^{6N}Q + 1,
$$

i.e.,

$$
\left\lfloor 2^{6N}\frac{x}{d} \right\rfloor = 2^{6N}Q,
$$

and $\mathit{cin} = 1$. Thus,

$$
\mathit{Q01} = \mathit{Q1} \equiv \mathit{QP} - \mathit{QN} \equiv 2^{6N}Q = \left\lfloor 2^{6N}\frac{x}{d} \right\rfloor \pmod{2^{6N}}
$$

and

$$
\mathit{Q01inc} = \mathit{Q1inc} \equiv \mathit{QP} - \mathit{QN} + \mathit{inc} \equiv 2^{6N}Q + \mathit{inc} = \left\lfloor 2^{6N}\frac{x}{d} \right\rfloor + \mathit{inc} \pmod{2^{6N}}.
$$

But if $R < 0$, then

$$2^{6N}Q - 1 < 2^{6N}\frac{x}{d} < 2^{6N}Q,$$

$$\left\lfloor 2^{6N}\frac{x}{d}\right\rfloor = 2^{6N}Q - 1,$$

$cin = 0$, and we have the same result:

$$Q01 = Q0 \equiv QP - QN - 1 \equiv 2^{6N}Q - 1 = \left\lfloor 2^{6N}\frac{x}{d}\right\rfloor \pmod{2^{6N}}$$

and

$$\begin{aligned} Q01inc = Q0inc &\equiv QP - QN - 1 + inc \\ &\equiv 2^{6N}Q - 1 + inc \\ &= \left\lfloor 2^{6N}\frac{x}{d}\right\rfloor + inc \pmod{2^{6N}}. \end{aligned}$$

Finally, we consider the data format. If $fmt = SP$, then $6N = 24 = p$, $inc = 2$,

$$Qtrunc = Q01 \equiv \left\lfloor 2^{6N}\frac{x}{d}\right\rfloor \pmod{2^p},$$

and

$$Qinc = Q01inc \equiv \left\lfloor 2^{6N}\frac{x}{d}\right\rfloor + 2 \pmod{2^p}.$$

If $fmt \neq SP$, then $6N = p + 1$, $inc = 4$,

$$Qtrunc = \left\lfloor \frac{Q01}{2}\right\rfloor \equiv \left\lfloor \frac{\lfloor 2^{6N}\frac{x}{d}\rfloor}{2}\right\rfloor = \left\lfloor 2^{6N}\frac{x}{d}\right\rfloor \pmod{2^p},$$

and

$$Qinc = \left\lfloor \frac{Q01inc}{2}\right\rfloor \equiv \left\lfloor \frac{\lfloor 2^{6N}\frac{x}{d}\rfloor + 4}{2}\right\rfloor = \left\lfloor 2^{6N}\frac{x}{d}\right\rfloor + 2 \pmod{2^p}. \ \square$$

Lemma 18.9 $stk = 0 \Leftrightarrow 2^p \cdot \frac{x}{d} \in Z.$

Proof We may assume $divPow2 = 0$. As we observed in the proof of Lemma 18.8, $remZero = 1 \Leftrightarrow R = 0$, and therefore,

$$stk = 0 \Leftrightarrow R = 0 \Leftrightarrow 2^{6N}\frac{x}{d} = 2^{6N}Q \Leftrightarrow 2^{6N}\frac{x}{d} \in Z.$$

If $fmt = SP$, then $6N = p$ and there is nothing further to prove, but in the other cases, $6N = p + 1$, and we must show that if $2^{p+1} \cdot \frac{x}{d} \in Z$, then $2^p \frac{x}{d} \in Z$.

Suppose this implication does not hold. Then $2^{p+1} \cdot \frac{x}{d}$ must be an odd integer a and $2^{p+1}x = ad$. According to Eqs. (18.4) and (18.5), either $2^{p+1}sig(A) = a \cdot sig(B)$ or $2^{p+2}sig(A) = a \cdot sig(B)$. In either case, $a \cdot sig(B)$ is an even integer, which we may denote as $2c$. There exist integers k and b, $k \geq 0$ and b odd, such that $sig(B) = 2^{-k}b$. But then $ab = 2^{k+1}c$, where a and b are odd, a contradiction. □

Since the rounder operates on the absolute value of the sum, the rounding mode $\mathcal{R}$ must be adjusted. We define

$$\mathcal{R}' = \begin{cases} RDN \text{ if } \mathcal{R} = RUP \text{ and } \frac{A}{B} < 0 \\ RUP \text{ if } \mathcal{R} = RDN \text{ and } \frac{A}{B} < 0 \\ \mathcal{R} \quad \text{otherwise.} \end{cases}$$

Lemma 18.10

(a) $\frac{x}{d}$ *is* p*-exact* $\Leftrightarrow inx = 0$*;*
(b) $\mathcal{R}'\left(2^p \frac{x}{d}, p\right) \equiv 2Qrnd[p-2:0] \pmod{2^p}$.

Proof We instantiate Lemma 6.104 with $z = 2^p \frac{x}{d}$ and $n = e = expo(z) = p$. Note that by Lemma 18.8, $Qtrunc[p-1:0] = \lfloor 2^p \frac{x}{d} \rfloor [p-1:0]$.

(a) By Lemma 18.9, $stk = 0 \Leftrightarrow 2^p \frac{x}{d} \in \mathbb{Z}$. Thus, by Lemma 6.104 (a) and the definition of *rounder*,

$$2^p \frac{x}{d} \text{ is } p\text{-exact} \Leftrightarrow Qtrunc[0] = \left\lfloor 2^p \frac{x}{d} \right\rfloor [0] = stk = 0 \Leftrightarrow inx = 0.$$

(b) First suppose that none of the conditions listed in Lemma 6.104 (b) holds. It is clear from the definition of *rounder* that in this case, $Qrnd = Qtrunc[53:1]$. Thus,

$$\begin{aligned} \mathcal{R}'\left(2^p \frac{x}{d}, p\right) &= RTZ\left(\left\lfloor 2^p \frac{x}{d} \right\rfloor, p\right) \\ &= 2\left\lfloor 2^p \frac{x}{d} \right\rfloor [p:1] \\ &\equiv 2\left\lfloor 2^p \frac{x}{d} \right\rfloor [p-1:1] \\ &= 2Qtrunc[p-1:1] \\ &= 2Qrnd[p-2:0] \pmod{2^p}. \end{aligned}$$

In the remaining case, $Qrnd = Qinc[53:1]$. Furthermore, either $stk = 1$ or $Qtrunc[0] = 1$, which implies $divPow2 = 0$, and therefore Lemma 18.8 applies.

Thus,

$$\begin{aligned}\mathcal{R}'\left(2^p\frac{x}{d}, p\right) &= fp^+\left(RTZ\left(\left\lfloor 2^p\frac{x}{d}\right\rfloor, p\right), p\right)\\ &= RTZ\left(\left\lfloor 2^p\frac{x}{d}\right\rfloor, p\right) + 2\\ &= 2\left\lfloor 2^p\frac{x}{d}\right\rfloor[p:1] + 2\\ &\equiv \left\lfloor 2^p\frac{x}{d}\right\rfloor - \left\lfloor 2^p\frac{x}{d}\right\rfloor[0] + 2\\ &\equiv Qinc - Qinc[0]\\ &\equiv 2Qinc[p-1:1]\\ &= 2Qrnd[p-2:0] \pmod{2^p}. \ \square\end{aligned}$$

The design is simplified by the observation that the quotient is never rounded up to a power of 2:

Lemma 18.11 $\mathcal{R}'\left(2^p\frac{x}{d}, p\right) \leq 2^{p+1} - 2.$

Proof It will suffice to show that $2^p\frac{x}{d} \leq 2^{p+1} - 2$, or $\frac{x}{d} \leq 2 - 2^{1-p}$.

If $sig(A) \geq sig(B)$, then by Eqs. 18.4 and 18.5,

$$\frac{x}{d} = \frac{sig(A)}{sig(B)} \leq \frac{2 - 2^{1-p}}{1} = 2 - 2^{1-p}.$$

If $sig(A) < sig(B)$, then

$$\frac{x}{d} = \frac{2sig(A)}{sig(B)} \leq \frac{2(sig(B) - 2^{1-p})}{sig(B)} = 2 - \frac{2^{2-p}}{sig(B)} < 2 - 2^{1-p}. \ \square$$

We consider the normal and subnormal cases separately:

Lemma 18.12 *If* $\left|\frac{A}{B}\right| \geq spn(fmt)$, *then*

(a) $\mathcal{R}'\left(\left|\frac{A}{B}\right|, p\right) = \left|\frac{A}{B}\right| \Leftrightarrow inx = 0$;
(b) $\mathcal{R}'\left(\left|\frac{A}{B}\right|, p\right) = 2^{expQ-bias-(p-1)}(2^{p-1} + Qrnd[p-2:0])$.

Proof

(a) This follows from Lemma 18.10 (a):

$$\mathcal{R}'\left(\left|\frac{A}{B}\right|, p\right) = \left|\frac{A}{B}\right| \Leftrightarrow \left|\frac{A}{B}\right| \text{ is } p\text{-exact} \Leftrightarrow \frac{x}{d} \text{ is } p\text{-exact} \Leftrightarrow inx = 0.$$

(b) By Lemma 18.11, $expo\left(\mathcal{R}'\left(2^p\frac{x}{d}, p\right)\right) = p$, and by Lemma 18.10 (b),

$$\mathcal{R}'\left(2^p\frac{x}{d}, p\right) = 2^p + \mathcal{R}'\left(2^p\frac{x}{d}, p\right) \bmod 2^p = 2^p + 2Qrnd[p-2:0]$$

and by Lemma 18.3 (e),

$$\mathcal{R}'\left(\left|\frac{A}{B}\right|, p\right) = 2^{expQ-bias-p}\mathcal{R}'\left(2^p\frac{x}{d}, p\right)$$
$$= 2^{expQ-bias-(p-1)}(2^{p-1} + Qrnd[p-2:0]). \; \square$$

For the subnormal case, we appeal to Lemmas 6.104 and 6.108:

Lemma 18.13 *If* $\left|\frac{A}{B}\right| < spn(fmt)$, *then*

(a) $drnd\left(\left|\frac{A}{B}\right|, \mathcal{R}', fmt\right) = \left|\frac{A}{B}\right| \Leftrightarrow inxDen = 0$;
(b) $drnd\left(\left|\frac{A}{B}\right|, \mathcal{R}', fmt\right) = 2^{1-bias-(p-1)}QrndDen[p-1:0]$.

Proof According to the definition of *rounder*,

$$QDen = 2^p + \left\lfloor 2^p\frac{x}{d}\right\rfloor \bmod 2^p = \left\lfloor 2^p\frac{x}{d}\right\rfloor,$$

$$Qshft = \lfloor 2^{-shft}QDen\rfloor = \left\lfloor 2^{p-shft}\frac{x}{d}\right\rfloor,$$

and

$$stkDen = 0 \Leftrightarrow Qshft = 2^{-shft}QDen \text{ and } QDen = 2^p\frac{x}{d}$$
$$\Leftrightarrow Qshft = 2^{p-shft}\frac{x}{d}.$$

By Lemma 18.3 (e),

$$expo\left(\frac{A}{B}\right) = expQ - bias < expo(spn(fmt)) = 1 - bias,$$

and hence $expQ \leq 0$.
Case 1: $expQ > 1 - p$.
In this case, $shft = 1 - expQ < p$. We shall invoke Lemma 6.104 with

$$n = p - shft = p + expQ - 1 > 0$$

and

$$z = 2^{p-shft}\frac{x}{d} = 2^{p-(1-expQ)}\frac{x}{d} = 2^n\frac{x}{d}.$$

Note that $Qshft = \lfloor z\rfloor$. By Definition 6.10,

$$drnd\left(\left|\frac{A}{B}\right|, \mathcal{R}', fmt\right) = \mathcal{R}'\left(\left|\frac{A}{B}\right|, p + expo\left(\frac{A}{B}\right) - expo(spn(fmt))\right)$$

$$= \mathcal{R}'\left(\left|\frac{A}{B}\right|, p + (expQ - bias) - (1 - bias)\right)$$
$$= \mathcal{R}'\left(\left|\frac{A}{B}\right|, n\right).$$

(a) This now follows from Lemmas 6.104 (a) and 18.3 (e):

$$drnd\left(\left|\frac{A}{B}\right|, \mathcal{R}', fmt\right) = \left|\frac{A}{B}\right| \Leftrightarrow \mathcal{R}'\left(\left|\frac{A}{B}\right|, n\right) = \left|\frac{A}{B}\right|$$
$$\Leftrightarrow 2^n\frac{x}{d} \in \mathbb{Z} \text{ and } Qshft[0] = \left\lfloor 2^n\frac{x}{d}\right\rfloor[0] = 0$$
$$\Leftrightarrow stkDen = grdDen = 0$$
$$\Leftrightarrow inxDen = 0.$$

(b) If none of the conditions listed in Lemma 6.104 (b) holds, then it is clear from the definition of *rounder* that $QrndDen = Qshft[53:1]$, and therefore

$$\mathcal{R}'\left(2^n\frac{x}{d}, n\right) = RTZ(Qshft, n) = 2Qshft[n:1] = 2QrndDen[p-1:0].$$

Otherwise, $QrndDen = Qshft[53:1] + 1$ and

$$\mathcal{R}'\left(2^n\frac{x}{d}, n\right) = fp^+(RTZ(Qshft, n), n)$$
$$= RTZ(Qshft, n) + 2$$
$$= 2(Qshft[n:1] + 1)$$
$$= 2QrndDen[p-1:0].$$

Thus, by Lemma 18.3,

$$\mathcal{R}'\left(\left|\frac{A}{B}\right|, n\right) = 2^{expQ-bias-n}\mathcal{R}'\left(2^n\frac{x}{d}, n\right)$$
$$= 2^{expQ-bias-(p+expQ-1)}2QrndDen[p-1:0]$$
$$= 2^{1-bias-(p-1)}QrndDen[p-1:0].$$

Case 2: $expQ \leq 1 - p$.

In this case, $p - shft \leq 0$, and therefore $Qshft < 2$. By Lemma 18.3,

$$\left|\frac{A}{B}\right| < 2^{expQ-bias+1} \leq 2^{2-p-bias} = spd(fmt).$$

(a) Either $drnd\left(\left|\frac{A}{B}\right|, \mathcal{R}', fmt\right) \geq spd(fmt)$ or $drnd\left(\left|\frac{A}{B}\right|, \mathcal{R}', fmt\right) = 0$. Thus,

$$drnd\left(\left|\frac{A}{B}\right|, \mathcal{R}', fmt\right) \neq \left|\frac{A}{B}\right|$$

and we must show $inxDen = 1$.

If $inxDen = 0$, then $stkDen = Qshft[0] = 0$. But if $stkDen = 0$, then as noted above, $Qshft = 2^{p-shft}\frac{x}{d} \neq 0$, which implies $Qshft = Qshft[0] = 1$.

(b) Since $Qshft \leq 1$, $lsbDen = Qshft[1] = 0$ and

$$\begin{aligned} grdDen = Qshft[0] = 1 &\Leftrightarrow Qshft = \left\lfloor 2^{p-shft}\frac{x}{d}\right\rfloor \geq 1 \\ &\Leftrightarrow p - shft = 0 \\ &\Leftrightarrow \left|\frac{A}{B}\right| \geq \frac{1}{2}spd(fmt). \end{aligned}$$

We invoke Lemma 6.108, considering the three cases of the lemma separately.

Suppose, for example, $\left|\frac{A}{B}\right| > \frac{1}{2}spd(fmt)$. Then $grdDen = stkDen = 1$. If $\mathcal{R}' = RNE$ or $\mathcal{R}' = RUP$, then by Lemma 6.108 (c), $drnd\left(\left|\frac{A}{B}\right|, \mathcal{R}', fmt\right) = spd(fmt)$, and according to the definition of *round*, $QrndDen = Qshft[53:1] + 1 = 1$, which implies

$$2^{1-bias-(p-1)}QrndDen[p-1:0] = 2^{1-bias-(p-1)} = spd(fmt)$$

as well. But if $\mathcal{R}' = RTZ$ or $\mathcal{R}' = RDN$, then $drnd\left(\left|\frac{A}{B}\right|, \mathcal{R}', fmt\right) = 0$, $QrndDen = Qshft[53:1] = 0$, and the claim again holds.

The other two cases are similar. □

Our correctness theorem for division is similar to that of multiplication (Theorem 16.1), matching the behavior of the top-level function *fdiv64* with the specification function *arm-binary-spec*. The proof is an extensive but entirely straightforward case analysis involving nothing more than inspection of the two functions and Lemmas 18.12 and 18.13.

Theorem 18.1 *Let*

$$\langle D_{spec}, R_{spec}\rangle = arm\text{-}binary\text{-}spec(DIV, opa, opb, R_{in}, fmt),$$

where $fmt \in \{DP, SP, HP\}$, opa and opb are 64-bit vectors, and R_{in} is a 32-bit vector, and let

$$\langle D, flags\rangle = fdiv64(opa, opb, fmt, R_{in}[24], R_{in}[25], R_{in}[23:22]).$$

Then $D = D_{spec}$ and $R_{in} \mid flags = R_{spec}$.

Chapter 19
Multi-Precision Radix-4 SRT Square Root

Finally, we present the function `fsqrt64`, which performs double-, single-, and half-precision square root extraction. This function, which is listed in Appendix E, is based on an implementation of the minimally redundant radix-4 case of SRT square root extraction characterized by Lemma 10.15 of Sect. 10.5. As noted in Chap. 18, it is derived from the same RTL module as the function `fdiv64`. The design shares hardware between the two operations for post-processing; therefore, the auxiliary functions `computeQ`, `rounder`, and `final` are shared by the two models.

The iterative phases, on the other hand, are implemented separately. Since the computation that updates the partial remainder is more complicated for the square root, two iterations of this algorithm rather than three are performed on each clock cycle. The resulting timing constraints do not require different approximations for the iterations within a cycle. This allows a simplification of the structure of the model: an iteration of the main `for` loop of `fsqrt64` corresponds to a single iteration of the algorithm rather than a cycle. Since the first iteration of the algorithm is once again executed before the loop is entered and the loop variable j ranges from 1 to $N - 1$, the number of iterations of the algorithm is N.

Notation The notational conventions established in Chap. 18 remain in force. Thus, for a variable that is assigned values within the main `for` loop, we shall use the subscript j to denote its value after $j-1$ iterations of the loop, i.e., after j iterations of the algorithm. When the subscript is omitted from a loop variable, it is understood to be N, corresponding to the final value. We shall similarly abbreviate the final quotient Q_N and remainder R_N as Q and R.

19.1 Pre-processing

The input and output parameters of `fsqrt64` are the same as those of `fdiv64`, except that the second operand, `opb`, is not present. Once again, the initial phase

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1_19

of this function handles the trivial early termination cases, in which the operand is a zero, a NaN, an infinity, a negative value, or a power of 2. We shall focus on the remaining case, in which the operand is either a positive denormal that is not forced to 0 or a positive normal. The operand value A, precision p, exponent width e, exponent bias $bias$, and rounding mode $\mathcal{R}$ are defined as in Sect. 18.1.

The function *normalize* performs a mantissa shift in the case of a denormal operand and returns values that satisfy the following. Note that while tighter bounds on the root exponent *expQ* could be achieved, those given are sufficient to preclude overflow and underflow:

Lemma 19.1

(a) $2^{52} \leq siga < 2^{53}$ *and* $siga[52 - p : 0] = 0$*;*
(b) $A = 2^{expShft-bias-52}siga$*;*
(c) $expQ = \left\lfloor \frac{expShft+bias}{2} \right\rfloor$ *and* $0 < expQ < 2^e - 2$.

Proof This is easily proved by inspection of the definition of *normalize*. □

We define

$$x = \begin{cases} \frac{sig(A)}{4} & \text{if } expShft \text{ is odd} \\ \frac{sig(A)}{2} & \text{if } expShft \text{ is even.} \end{cases}$$

Clearly $\frac{1}{4} \leq x < 1$, as required by the algorithm of Sect. 10.5. Since we have defined x and q_j for $1 \leq j \leq N$, the definitions of the partial roots and remainders Q_j and R_j for $0 \leq j \leq N$ are given by Eqs. (10.13) and (10.14).

The algorithm computes an approximation of $\sqrt{x}$, which is related to the desired final result $\sqrt{A}$ according to the following:

Lemma 19.2 $A = 2^{2(expQ-bias+1)}x$.

Proof If *expShft* is odd, then

$$expQ = \frac{expShft + bias}{2} = \frac{expShft - bias}{2} + bias$$

and

$$A = (2^{expShft-bias+2})(2^{-54}siga) = 2^{2\left(\frac{expShft-bias}{2}+1\right)}x = 2^{2(expQ-bias+1)}x.$$

If *expShft* is even, then

$$expQ = \frac{expShft + bias - 1}{2} = \frac{expShft - bias + 1}{2} + bias - 1$$

and

$$A = (2^{expShft-bias+1})(2^{-53}siga) = 2^{2\left(\frac{expShft-bias+1}{2}\right)}x = 2^{2(expQ-bias+1)}x.$$

□

The remaining iterations are performed within the `for` loop by the functions *nextDigit*, *nextRem*, and *nextRoot*.

The first iteration is performed by the function *firstIter*, the values of which satisfy the following:

Lemma 19.3

(a) q_1 *is the greatest* $k \in \{-2, \dots, 2\}$ *such that* $m_k(8,0) \leq A_0$*;*
(b) $2^{54}(Q_1 - 1) = QP_1 - QN_1$ *and* $QP_1[51:0] = QN_1[51:0] = 0$*;*
(c) $2^{55}R_1 = RP_1 - RN_1$ *and* $RP_1[52-p:0] = RN_1[52-p:0] = 0$*;*

Proof We shall consider the case in which *expShft* is odd and $siga[51] = 1$; the other three cases are similar.

(a) Since $q_1 = -1$, we must show that $m_{-1}(8,0) \leq 4R_0 < m_0(8,0)$, where $R_0 = x - 1$. But since $2^{52} + 2^{51} \leq siga < 2^{53}$ and $x = 2^{-54}siga$, $\frac{3}{8} \leq x < \frac{1}{2}$ and

$$m_{-1}(8,0) = -\frac{5}{2} \leq 4(x-1) < -2 < -1 = m_0(8,0).$$

(b) $Q_1 = 1 + 4^{-1}(-1) = \frac{3}{4}$ and $QP_1 - QN_1 = 0 - 2^{52} = 2^{54}(\frac{3}{4} - 1)$.
(c) $RP_1 = 2^3 siga + 2^{58} + 2^{57} = 2^{59} + 2^{57}(x-1) = 2^{57}R_0$ and $RN_1 = 2^{53} + 2^{59} - 2^{56} = 2^{59} - 2^{53}7$. By Lemma 19.1 (a), $RP_1[52-p:0] = RN_1[52-p:0] = 0$, and by (10.14),

$$R_1 = 4R_0 - (-1)(2(1) + 4^{-1}(-1)) = 4R_0 - \frac{7}{4} = 2^{-55}(RP_1 - RN_1).$$

□

19.2 Iterative Phase

The remainder approximation A_j of Lemma 10.15 is defined by

$$A_j = \begin{cases} 4R_0 & \text{if } j = 0 \\ \frac{1}{8}si(RS7_j, 7) & \text{if } 1 \leq j \leq N. \end{cases}$$

A_j and R_j are related according to the comparison constants of Fig. 10.3 of Sect. 10.4 as specified by the following lemma:

Lemma 19.4 *The following conditions hold for all j, $1 \leq j \leq N$:*

(a) If $j' = \min(j-1, 2)$ and $i = 16\left(Q_{j'} - \frac{1}{2}\right)$, then q_j is the greatest $k \in \{-2, \ldots, 2\}$ such that $m_k(i, j-1) \leq A_{j-1}$;
(b) For all $k \in \{-1, \ldots, 2\}$,

$$A_{j-1} < m_k(i, j-1) \Rightarrow 4R_{j-1} < m_k(i, j-1)$$

and

$$A_{j-1} \geq m_k(i, j-1) \Rightarrow 4R_{j-1} > m_k(i, j-1) - \frac{1}{32};$$

(c) $\underline{B}(j) \leq R_j \leq \overline{B}(j)$;
(d) $\frac{1}{2} \leq Q_j \leq 1$;
(e) $2^{54}(Q_j - 1) = QP_j - QN_j$ and $QP_j[53-2j:0] = QN_j[53-2j:0] = 0$;
(f) $2^{55}R_j \in \mathbb{Z}$, $2^{55}R_j \equiv RP_j - RN_j \pmod{2^{59}}$ and $RP_j[52-p:0] = RN_j[52-p:0] = 0$.

Proof We first consider the case $j = 1$. We have $Q_{j'} = Q_0 = 1$ and $i = 8$. Since $A_0 = 4R_0$, (b) holds trivially and (a), (e), and (f) correspond to Lemma 19.3 (a), (b), and (c); (c) and (d) then follow from Lemma 10.15.

Let $1 \leq j < N$ and assume that the lemma holds for all ℓ, $1 \leq \ell \leq j$. We shall show that all claims hold for $j+1$:

(a) First note that the value i_j computed by *fsqrt64* coincides with the value i defined above: If $j = 1$, then this is clear from the definition of *firstIter*, and if $j \geq 2$, then

$$i_j = i_2 = i_1 + q_2 = 16\left(Q_1 - \frac{1}{2}\right) + q_2 = 16\left(Q_1 + 4^{-2}q_2 - \frac{1}{2}\right) = 16\left(Q_2 - \frac{1}{2}\right).$$

Now consider the computation of $nextDigit(RP_j, RN_j, i, j)$.

Recall that $A_j = \frac{1}{8}si(RS7, 7)$. It is clear from the definition of *nextDigit* that q_{j+1} is the greatest k such that $8m_k(i, j) \leq si(RS7, 7)$, or $m_k(i, j) \leq A_j$.

(b) We must show that for all k,

$$A_j < m_k(i, j) \Rightarrow 4R_j < m_k(i, j)$$

and

$$A_j \geq m_k(i, j) \Rightarrow 4R_j > m_k(i, j) - \frac{1}{32}.$$

We shall invoke Lemma 2.54 with

$$X = RP4 - RN4 \equiv 4(RP_j - RN_j) \equiv 4(2^{55} R_j) = 2^{57} R_j \pmod{2^{59}}$$

and

$$Y = 2^{50} y,$$

where

$$y = RP4[58:50] - RN4[58:50].$$

Let $\bar{X} = X \bmod 2^{59}$, $\bar{Y} = Y \bmod 2^{59}$, and $\bar{y} = y \bmod 2^9$. By Lemmas 1.18 and 2.52,

$$si(\bar{Y}, 59) = si(2^{50}\bar{y}, 59) = 2^{50} si(\bar{y}, 9).$$

Since

$$|R_j| \leq \overline{B}(j) = 2 \cdot \frac{2}{3} Q_j + \left(\frac{2}{3}\right)^2 4^{-j} \leq \frac{4}{3} + \frac{1}{9} < 2 - 2^{-7},$$

$|2^{57} R_j| < 2^{58} - 2^{50}$, which implies $2^{57} R_j = si(\bar{X}, 59)$. Thus, the hypothesis of the lemma is satisfied and we may conclude that

$$\begin{aligned}
|2^{57} R_j - 2^{50} si(\bar{y}, 9)| &= |si(\bar{X}, 59) - si(\bar{Y}, 59)| \\
&= |X - Y| \\
&= |RP4[49:0] - RN4[49:0]| \\
&< 2^{50}.
\end{aligned}$$

It is clear by a case analysis on $RP4[50]$ and $RN4[50]$ that $RS8[7:0] = \bar{y}[8:1]$, and it follows that $RS7 = \bar{y}[8:2]$. Consequently, for $m \in \mathbb{Z}$,

$$si(RS7, 7) \geq m \Leftrightarrow si(\bar{y}, 9) \geq 4m.$$

Thus,

$$\begin{aligned}
A_j < m_k(i, j) &\Rightarrow si(RS7, 7) < 8m_k(i, j) \\
&\Rightarrow si(\bar{y}, 9) < 32m_k(i, j) \\
&\Rightarrow si(\bar{y}, 9) \leq 32m_k(i, j) - 1 \\
&\Rightarrow 2^{57} R_j < 2^{50}(32m_k(i, j) - 1) + 2^{50} = 2^{55} m_k(i, j) \\
&\Rightarrow 4R_j < m_k(i, j).
\end{aligned}$$

Similarly,

$$\begin{aligned} A_j \geq m_k(i,j) &\Rightarrow si(RS7, 7) \geq 8m_k(i,j) \\ &\Rightarrow si(\bar{y}, 9) \geq 32m_k(i,j) \\ &\Rightarrow 2^{57}R_j > 2^{50}32m_k(i,j) - 2^{50} = 2^{55}\left(m_k(i,j) - \frac{1}{32}\right) \\ &\Rightarrow 4R_j > m_k(i,j) - \frac{1}{32}. \end{aligned}$$

(c) This follows from Lemma 10.15, as does (d).
(e) This follows from Eq. (10.13) and the definition of *nextRoot*.
(f) Consider the computation of $nextRem(RP_j, RN_j, QP_j, QN_j, q_{j+1}, j, fmt)$.
Let $D = 2Q_j + 4^{-(j+1)}q_{j+1}$. Then (10.14) may be written as $R_{j+1} = 4R_j - q_{j+1}D$. Clearly,

$$\begin{aligned} Dcar - Dsum &= 2^{56} + 4QP_j - 4QN_j + 2^{53-2j}q_{j+1} \\ &= 2^{56} + 2^{56}(Q_j - 1) + 2^{55}2^{-2-2j}q_{j+1} \\ &= 2^{55}\left(2Q_j + 4^{-(j+1)}q_{j+1}\right) \\ &= 2^{55}D, \end{aligned}$$

and

$$RP4 - RP4 \equiv 4(RP_j - RN_j) \equiv 2^{55}4R_j \pmod{2^{59}}.$$

If $q_{j+1} = 0$, then the claim holds trivially:

$$RP_{j+1} - RN_{j+1} = RP4 - RP4 \equiv 2^{55}4R_j = 2^{55}R_{j+1} \pmod{2^{59}}.$$

Suppose $q_{j+1} \neq 0$. Then

$$DQcar - DQsum = -q_{j+1}(Dcar - Dsum) = -2^{55}q_{j+1}D.$$

As a notational convenience, for a 59-bit vector V, let $V' = V[58:53-p]$. Note that if $V[52-p:0] = 0$, then $V' = 2^{p-53}V$. Clearly, this holds for the vectors *DQcar*, *DQsum*, *RP4*, and *RN4*.

Now according to the definitions of *sum1* and *car1*,

$$\tilde{}sum1' = \tilde{}RN4' \wedge RP4' \wedge DQcar'$$

and

$$car1' = 2(\tilde{}RN4' \,\&\, RP4' \mid (\tilde{}RN4' \mid RP4') \,\&\, DQcar'),$$

which implies $car1' + \tilde{~}sum1' = RP4' + \tilde{~}RN4' + DQcar'$, and

$$\begin{aligned} car1' - sum1' &\equiv car1' + \tilde{~}sum1' + 1 \\ &\equiv RP4' + \tilde{~}RN4' + DQcar' + 1 \\ &\equiv RP4' - RN4' + DQcar' \pmod{2^{p+6}}. \end{aligned}$$

Similarly,

$$\begin{aligned} car2' + \tilde{~}sum2' &\equiv \tilde{~}sum1' + car1' + \tilde{~}DQsum' + 1 \\ &\equiv \tilde{~}sum1' + car1' - DQsum' \pmod{2^{p+6}} \end{aligned}$$

and

$$\begin{aligned} car2' - sum2' &\equiv car2' + \tilde{~}sum2' + 1 \\ &\equiv \tilde{~}sum1' + car1' - DQsum' + 1 \\ &\equiv car1' - sum1' - DQsum' \\ &\equiv (RP4' - RN4') + (DQcar' - DQsum') \\ &\equiv 2^{p-53}2^{55}4R_j - 2^{55}q_{j+1}D \\ &= 2^{p+2}R_{j+1} \pmod{2^{p+6}}. \end{aligned}$$

But

$$RP_{j+1} - RN_{j+1} = 2^{53-p}car2' - 2^{53-p}sum2' \equiv 2^{55}R_{j+1} \pmod{2^{59}}.$$

□

19.3 Post-Processing and Rounding

In comparison to division, post-processing of the square root is simplified by the absence of underflow and overflow (Lemma 19.1 (c)), but has the minor complication that the result may round up to a power of 2.

On the other hand, the formal proof of correctness is significantly complicated by the limitations of the ACL2 logic, in which the square root function cannot be explicitly defined. Thus, the ACL2 specification of this operation, instead of referring directly to the desired rounded result $\mathcal{R}(\sqrt{A}, p)$, is expressed in terms $\mathcal{R}(\sqrt[(p+2)]{A}, p)$, where $\sqrt[(p+2)]{A}$ is the $(p+2)$-bit approximation of the square root discussed in Chap. 7.

However, in order to establish the required bound on the approximation error of the final root Q, it will be necessary to base our analysis instead on the value $\sqrt[(2N+1)]{A}$, with $(2N+1)$-bit accuracy, where $2N+1$ is either $p+2$ or $p+3$ depending on the data format. This is justified by Lemma 7.16, which ensures that

$$\mathcal{R}(\sqrt[(p+2)]{A}, p) = \mathcal{R}(\sqrt[(2N+1)]{A}, p).$$

The rounder actually produces a rounding of $\sqrt[(2N+1)]{x}$, which is related to the desired result as follows:

Lemma 19.5 $\sqrt[(2N+1)]{A} = 2^{expQ-bias+1} \sqrt[(2N+1)]{x}$.

Proof This follows from Lemmas 19.2 and 7.15. □

Lemma 19.6

(a) $(Q - \frac{2}{3}4^{-N})^2 \leq x \leq (Q + \frac{2}{3}4^{-N})^2$;
(b) $|\sqrt[(2N+1)]{x} - Q| < 2^{-2N}$.

Proof (a) is a consequence of Lemmas 19.4 (c) and 10.10. It follows that

$$(Q - 4^{-N})^2 < x < (Q + 4^{-N})^2,$$

and (b) will follow from Lemma 7.18 once we show that $Q - 4^{-N}$ and $Q + 4^{-N}$ are both $2N$-exact.

Since $Q - 4^{-N} < 1$, $expo(Q - 4^{-N}) \leq -1$ and since $4^N Q \in \mathbb{Z}$,

$$2^{2N-1-expo(Q-4^{-N})}(Q - 4^{-N}) = 2^{-1-expo(Q-4^{-N})}4^{-N}(Q - 4^{-N}) \in \mathbb{Z},$$

i.e., $Q - 4^{-N}$ is $2N$-exact. To draw the same conclusion about $Q + 4^{-N}$, it will suffice to show that $Q + 4^{-N} \leq 1$.

Since $x < 1$ and x is p-exact, we have $(Q - 4^{-N})^2 < x \leq 1 - 2^{-p} < (1 - 2^{-p-1})^2$. Thus, $Q - 4^{-N} < 1 - 2^{-p-1}$ and $Q < 1 - 2^{-p-1} + 2^{-2N} \leq 1$. It follows that $Q \leq 1 - 4^{-N}$, i.e., $Q + 4^{-N} \leq 1$. □

Lemma 19.7 $\sqrt[(2N+1)]{x} < 1 - 2^{-p-1}$ *and* $Q \leq 1 - 2^{-p-1}$.

Proof As we have noted, $x < (1 - 2^{-p-1})^2$. The bound on $\sqrt[(2N+1)]{x}$ follows from Lemma 7.18. To establish the bound on Q, first suppose $fmt \neq SP$. Then $2N = p+1$ and by Lemma 19.6,

$$Q < \sqrt[(2N+1)]{x} + 2^{-p-1} < 1,$$

and hence $Q \leq 1 - 2^{-2N} = 1 - 2^{-p-1}$. On the other hand, if $fmt = SP$, then $2N = p + 2$ and

$$Q < \sqrt[(2N+1)]{x} + 2^{-p-2} < 1 - 2^{-p-1} + 2^{-p-2} = 1 - 2^{-p-2},$$

which again implies $Q \leq 1 - 2^{-p-1}$. □

Once again, the rounder selects one of two values that are returned by *computeQ*:

Lemma 19.8

(a) $Qtrunc \equiv \lfloor 2^{p+1}\sqrt[(2N+1)]{x} \rfloor \pmod{2^p}$;
(b) $Qinc \equiv \lfloor 2^{p+1}\sqrt[(2N+1)]{x} \rfloor + 2 \pmod{2^p}$.

Proof The proof closely follows that of Lemma 18.8. Equations (18.6)–(18.9) hold as in the division case, and once again, the sign of the final remainder is used to select either *Q1* and *Q1inc* or *Q0* and *Q0inc*.

It follows from Lemma 19.4 that $rem = 2^{56}R \bmod 2^{59}$, and therefore, by (10.14) and Lemma 7.18,

$$remSign = 0 \Leftrightarrow R \geq 0 \Leftrightarrow \sqrt[(2N+1)]{x} \geq Q$$

and

$$remZero = 1 \Leftrightarrow R = 0 \Leftrightarrow \sqrt[(2N+1)]{x} = Q.$$

By Lemma 19.6, $|2^{2N}Q - 2^{2N}\sqrt[(2N+1)]{x}| < 1$. If $R \geq 0$, then

$$2^{2N}Q \leq 2^{2N}\sqrt[(2N+1)]{x} < 2^{2N}Q + 1,$$

i.e.,

$$\lfloor 2^{2N}\sqrt[(2N+1)]{x} \rfloor = 2^{2N}Q,$$

and $cin = 1$. Thus,

$$Q01 = Q1 \equiv QP - QN \equiv 2^{2N}Q = \lfloor 2^{2N}\sqrt[(2N+1)]{x} \rfloor \pmod{2^{2N}}$$

and

$$\begin{aligned} Q01inc = Q1inc &\equiv QP - QN + inc \\ &\equiv 2^{2N}Q + inc \\ &= \lfloor 2^{2N}\sqrt[(2N+1)]{x} \rfloor + inc \pmod{2^{2N}}. \end{aligned}$$

But if $R < 0$, then

$$2^{2N}Q - 1 < 2^{2N}\sqrt[(2N+1)]{x} < 2^{2N}Q,$$

$$\lfloor 2^{2N}\sqrt[(2N+1)]{x} \rfloor = 2^{2N}Q - 1,$$

$cin = 0$, and we have the same result:

$$Q01 = Q0 \equiv QP - QN - 1 \equiv 2^{2N}Q - 1 = \lfloor 2^{2N}\sqrt[(2N+1)]{x} \rfloor \pmod{2^{2N}}$$

and

$$\begin{aligned} Q01inc = Q0inc &\equiv QP - QN - 1 + inc \\ &\equiv 2^{2N} Q - 1 + inc \\ &= \left\lfloor 2^{2N}\ \sqrt[(2N+1)]{x} \right\rfloor + inc \pmod{2^{2N}}. \end{aligned}$$

Finally, we consider the data format. If *fmt* $\neq SP$, then $2N = p + 1$, $inc = 2$,

$$Qtrunc \equiv Q01 \equiv \left\lfloor 2^{2N}\ \sqrt[(2N+1)]{x} \right\rfloor = \left\lfloor 2^{p+1}\ \sqrt[(2N+1)]{x} \right\rfloor \pmod{2^p},$$

and

$$Qinc \equiv Q01inc \equiv \left\lfloor 2^{2N}\ \sqrt[(2N+1)]{x} \right\rfloor + 2 = \left\lfloor 2^{p+1}\ \sqrt[(2N+1)]{x} \right\rfloor + 2 \pmod{2^p}.$$

If *fmt* $= SP$, then $2N = 26 = p + 2$, $inc = 4$,

$$\begin{aligned} Qtrunc \equiv \left\lfloor \frac{Q01}{2} \right\rfloor &\equiv \left\lfloor \frac{\left\lfloor 2^{2N}\ \sqrt[(2N+1)]{x} \right\rfloor}{2} \right\rfloor \\ &= \left\lfloor 2^{2N-1}\ \sqrt[(2N+1)]{x} \right\rfloor \\ &= \left\lfloor 2^{p+1}\ \sqrt[(2N+1)]{x} \right\rfloor \pmod{2^p}, \end{aligned}$$

and

$$\begin{aligned} Qinc \equiv \left\lfloor \frac{Q01inc}{2} \right\rfloor &\equiv \left\lfloor \frac{\left\lfloor 2^{2N}\ \sqrt[(2N+1)]{x} \right\rfloor + 4}{2} \right\rfloor \\ &= \left\lfloor 2^{2N-1}\ \sqrt[(2N+1)]{x} \right\rfloor + 2 \\ &= \left\lfloor 2^{p+1}\ \sqrt[(2N+1)]{x} \right\rfloor + 2 \pmod{2^p}. \end{aligned}$$

□

The function *computeQ* also returns an indication of inexactness:

Lemma 19.9 $stk = 0 \Leftrightarrow 2^{p+1}\ \sqrt[(2N+1)]{x} \in Z.$

Proof As observed above, $remZero = 1 \Leftrightarrow \sqrt[(2N+1)]{x} = Q$, and therefore,

$$stk = 0 \Leftrightarrow 2^{2N}\ \sqrt[(2N+1)]{x} = 2^{2N} Q \Leftrightarrow 2^{2N}\ \sqrt[(2N+1)]{x} \in Z.$$

Since $2N$ is either $p + 1$ or $p + 2$, it suffices to show that if $2N = p + 2$ and $2^{2N} \cdot \sqrt[(2N+1)]{x} \in Z$, then $2^{p+1}\ \sqrt[(2N+1)]{x} \in Z$.

Suppose this implication does not hold. Then $\sqrt[(2N+1)]{x}$ is $2N$-exact, and by Corollary 7.19, $(\sqrt[(2N+1)]{x})^2 = x$. Furthermore, $2^{2N} \cdot \sqrt[(2N+1)]{x}$ is an odd integer. But this implies that $(2^{2N} \cdot \sqrt[(2N+1)]{x})^2 = 2^{2p+4}x$ is an odd integer, which is impossible, since $2^{p+1}x$ is an integer. □

Lemma 19.10

(a) $\sqrt[(2N+1)]{x}$ *is* p*-exact* $\Leftrightarrow inx = 0$;
(b) $\mathcal{R}\left(2^{p+1}\sqrt[(2N+1)]{x}, p\right) \equiv 2Qrnd[p-2:0] \pmod{2^p}$.

Proof The proof may be derived from that of Lemma 18.10 simply by replacing $2^p\frac{x}{d}$ with $2^{p+1}\sqrt[(2N+1)]{x}$. □

It is not difficult to show that $\sqrt[(2N+1)]{x}$ rounds up to 1 only if the rounding mode is *RUP* and x has the maximum value $1 - 2^{-p}$. For timing reasons, however, the implementation instead makes this determination by observing that the sequence of root digits satisfies conditions that imply that the final root has its maximum value, $1-2^{-p-1}$. In this event, the variable *expInc* is set, causing *expQ* to be incremented:

Lemma 19.11 *The following are equivalent:*

(a) $\mathcal{R}(\sqrt[(2N+1)]{x}, p) = 1$;
(b) $\mathcal{R} = RUP$ *and* $x = 1 - 2^{-p}$;
(c) $\mathcal{R} = RUP$ *and* $Q = 1 - 2^{-p-1}$;
(d) $expInc = 1$.

Proof Since $\sqrt[(2N+1)]{x} < 1 - 2^{-p-1}$, $\mathcal{R}(\sqrt[(2N+1)]{x}, p) < 1$ unless $\mathcal{R} = RUP$.

If $x = 1 - 2^{-p}$, then $x > (1 - 2^{-p})^2$ and by Lemma 7.18, $\sqrt[(2N+1)]{x} > 1 - 2^{-p}$, which implies $RUP(\sqrt[(2N+1)]{x}, p) = 1$. Conversely, if $x \neq 1 - 2^{-p}$, then since x is p-exact, $x \leq 1 - 2^{1-p} < (1 - 2^{-p})^2$, which implies $\sqrt[(2N+1)]{x} < 1 - 2^{-p}$ and $RUP(\sqrt[(2N+1)]{x}, p) \leq 1 - 2^{-p}$.

If $Q = 1 - 2^{-p-1}$, then $\sqrt[(2N+1)]{x} > 1 - 2^{-p-1} - 2^{-2N} \geq 1 - 2^{-p}$ and $RUP(\sqrt[(2N+1)]{x}, p) = 1$. Otherwise, $Q < 1-2^{-p-1}$, which implies $Q \leq 1-2^{-p-1} - 2^{-2N}$ and by Lemma 19.6(a),

$$x \leq \left(Q + \frac{2}{3} \cdot 4^{-N}\right)^2 \leq \left(1 - 2^{-p-1} - \frac{1}{3} \cdot 2^{-2N}\right)^2 < 1 - 2^{-p}.$$

Regarding (d), it is clear that $expInc = 1$ iff the following conditions hold: *opa* is normal, $\mathcal{R} = RUP$ and

$$q_j = \begin{cases} 0 \text{ if } j < N \\ -1 \text{ if } j = N \text{ and } fmt \neq SP \\ -2 \text{ if } j = N \text{ and } fmt = SP. \end{cases}$$

Suppose $expInc = 1$. Then by (10.13) and the definition of N,

$$Q = \begin{cases} 1 - 4^{-N} = 1 - 2^{2N} = 1 - 2^{-p-1} & \text{if } fmt \neq SP \\ 1 - 2 \cdot 4^{-N} = 1 - 2^{1-2N} = 1 - 2^{-p-1} & \text{if } fmt = SP. \end{cases}$$

Conversely, suppose $RUP(\sqrt[(2N+1)]{x}, p) = 1$. Then *opa* must be normal, for if *opa* were denormal, then x would be $(p-1)$-exact, contradicting $x = 1 - 2^{-p}$.

We shall show by induction on j that $q_j = 0$ for $1 \leq j < N$. Let $0 \leq j < N - 1$ and assume that $q_1 = \ldots q_j = 0$. Then $Q_j = 1$,

$$R_j = 4^j(x - Q_j^2) = 4^j(1 - 2^{-p} - 1) = -2^{2j-p},$$

and

$$4R_j = -2^{2j+2-p} \geq -2^{2(N-2)+2-p} = -2^{2N-(p+2)} \geq -1 = m_0(8, j).$$

By Lemma 18.5(a), we also have $A_j \geq m_0(8, j)$, and consequently $q_{j+1} \geq 0$. But since $Q_{j+1} \leq 1$, this implies $q_{j+1} = 0$.

Thus, $Q_{N-1} = 1$ and $Q = 1 + 4^{-N}q_N = 1 - 2^{-p-1}$, which implies

$$q_N = -2^{2N-p-1} = \begin{cases} -1 \text{ if } fmt \neq SP \\ -2 \text{ if } fmt = SP. \end{cases}$$

□

We have an analog of Lemma 18.12, but since underflow cannot occur, we need no analog of Lemma 18.13:

Lemma 19.12

(a) $\mathcal{R}(\sqrt[(2N+1)]{A}, p) = \sqrt[(2N+1)]{A} \Leftrightarrow inx = 0$;
(b) $\mathcal{R}(\sqrt[(2N+1)]{A}, p) = 2^{expRnd-bias-(p-1)}(2^{p-1} + Qrnd[p-2:0])$.

Proof

(a) This follows from Lemmas 19.5 and 19.10 (a):

$$\begin{aligned}\mathcal{R}(\sqrt[(2N+1)]{A}, p) = \sqrt[(2N+1)]{A} &\Leftrightarrow \sqrt[(2N+1)]{A} \text{ is } p\text{-exact} \\ &\Leftrightarrow \sqrt[(2N+1)]{x} \text{ is } p\text{-exact} \\ &\Leftrightarrow inx = 0.\end{aligned}$$

(b) First suppose $\mathcal{R}(\sqrt[(2N+1)]{x}, p) < 1$. Then

$$expo(\mathcal{R}(2^{p+1}\sqrt[(2N+1)]{x}, p)) = expo(2^{p+1}\sqrt[(2N+1)]{x}) = p + 1 - 1 = p.$$

Therefore, by Lemma 19.10 (b),

$$\mathcal{R}(2^{p+1}\sqrt[(2N+1)]{x}, p) = 2^p + \mathcal{R}(2^{p+1}\sqrt[(2N+1)]{x}, p) \bmod 2^p = 2^p + 2Qrnd[p-2:0]$$

and by Lemmas 19.5 and 19.11,

$$\begin{aligned}\mathcal{R}(\sqrt[(2N+1)]{A}, p) &= 2^{expQ-bias+1-(p+1)}\mathcal{R}(2^{p+1}\sqrt[(2N+1)]{x}, p) \\ &= 2^{expRnd-bias+1-(p+1)}\mathcal{R}(2^{p+1}\sqrt[(2N+1)]{x}, p)\end{aligned}$$

$$= 2^{expRnd-bias-(p-1)}(2^{p-1} + Qrnd[p-2:0]).$$

In the remaining special case, $\mathcal{R}(\sqrt[(2N+1)]{x}, p) = 1$ and by Lemma 19.10 (b),

$$2Qrnd[p-2:0] = \mathcal{R}(2^{p+1}\sqrt[(2N+1)]{x}, p) \bmod 2^p = 2^{p+1} \bmod 2^p = 0.$$

According to Lemma 19.11, *expInc* = 1, and therefore

$$\begin{aligned}\sqrt[(2N+1)]{A} &= 2^{expQ-bias+1}\sqrt[(2N+1)]{x}\\ &= 2^{(expRnd-1)-bias+1}\sqrt[(2N+1)]{x}\\ &= 2^{expRnd-bias}\sqrt[(2N+1)]{x}\end{aligned}$$

and

$$\begin{aligned}\mathcal{R}(\sqrt[(2N+1)]{A}, p) &= 2^{expRnd-bias}\mathcal{R}(\sqrt[(2N+1)]{x}, p)\\ &= 2^{expRnd-bias-(p-1)}(2^{p-1} + Qrnd[p-2:0]).\end{aligned}$$

□

The statement and proof of our correctness theorem are quite similar to those of Theorem 18.1, although simplified by the absence of overflow and underflow:

Theorem 19.1 *Let*

$$\langle D_{spec}, R_{spec}\rangle = \textit{arm-sqrt-spec}(opa, R_{in}, fmt),$$

where opa is a 64-bit vector, R_{in} is a 32-bit vector, and fmt $\in \{DP, SP, HP\}$*, and let*

$$\langle D, flags\rangle = fsqrt64(opa, fmt, R_{in}[24], R_{in}[25], R_{in}[23:22]).$$

Then $D = D_{spec}$ and $R_{in} \mid flags = R_{spec}$.

Appendices

These appendices contain the pseudocode versions of the RTL models of Chaps. 16–19.

A Common Code

```
// This section contains constant declarations and utility functions that are
// shared by modules of subsequent sections.

// Formats:

enum Format {HP, SP, DP};

// Data classes:

enum Class {ZERO, INF, SNAN, QNAN, NORM, DENORM};

// Rounding modes:

const ui2 rmodeNear = 0, rmodeUP = 1, rmodeDN = 2 rmodeZero = 3;

// Flags:

const uint IDC = 7, IXC = 4, UFC = 3, OFC = 2, DZC = 1, IOC = 0;

// Extract operand components, apply FZ, identify data class,
// and record denormal exception:

<bool, ui11, ui52, Class, ui8> analyze(ui64 op, Format fmt, bool fz, ui8 flags){

  // Extract fields:
  bool sign;
  ui11 exp;
  ui52 man, manMSB;
  bool expIsMax;
  switch (fmt) {
  case DP:
```

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1

```
    sign = op[63];
    exp = op[62:52];
    expIsMax = exp == 0x7FF;
    man = op[51:0];
    manMSB = 0x8000000000000;
    break;
  case SP:
    sign = op[31];
    exp = op[30:23];
    expIsMax = exp == 0xFF;
    man = op[22:0];
    manMSB = 0x400000;
    break;
  case HP:
    sign = op[15];
    exp = op[14:10];
    expIsMax = exp == 0x1F;
    man = op[9:0];
    manMSB = 0x200;
  }

  // Classify:
  Class c;
  if (expIsMax) { // NaN or infinity
    if (man == 0) {
      c = INF;
    }
    else if (man & manMSB) {
      c = QNAN;
    }
    else  {
      c = SNAN;
    }
  }
  else if (exp == 0) { // zero or denormal
    if (man == 0) {
      c = ZERO;
    }
    else if (fz) {
      c = ZERO;
      if (fmt != HP) {
        flags[IDC] = 1; // denormal exception
      }
    }
    else  {
      c = DENORM;
    }
  }
  else  { // normal
    c = NORM;
  }
  return <sign, exp, man, c, flags>;
}

// Count leading zeroes of a nonzero 53-bit vector.
// After k iterations of the loop, where 0 <= k <= 6, the value of n is 2^(6-k)
// and the low n entries of z and c are as follows:
// Consider the partition of x into n bit slices of width 2^k. For 0 <= i < n,
// the i^th slice is x[2^k*(i+1)-1:2^k*i]. Let L(i) be the number of leading
// zeroes of this slice.  Then
//   z[i] = 1 <=> L(i) = 2^k;
//   L(i) < 2^k => c[i] = L(i).
```

```
ui6 CLZ53(ui53 m) {
  ui64 x = 0;
  x[63:11] = m;
  bool z[64];
  ui6 c[64];
  for (uint i = 0; i < 64; i++) {
    z[i] = !x[i];
    c[i] = 0;
  }
  uint n = 64;
  for (uint k = 0; k < 6; k++) {
    n = n / 2; // n = 2^(5-k)
    for (uint i = 0; i < n; i++) {
      c[i] = z[2 * i + 1] ? c[2 * i] : c[2 * i + 1];
      c[i][k] = z[2 * i + 1];
      z[i] = z[2 * i + 1] && z[2 * i];
    }
  }
  return c[0];
}

// A 128-bit version, used by fadd64:

ui7 CLZ128(ui128 x) {
  bool z[128];
  ui7 c[128];
  for (uint i = 0; i < 128; i++) {
    z[i] = !x[i];
    c[i] = 0;
  }
  uint n = 128;
  for (uint k = 0; k < 7; k++) {
    n = n / 2;
    for (uint i = 0; i < n; i++) {
      c[i] = z[2 * i + 1] ? c[2 * i] : c[2 * i + 1];
      c[i][k] = z[2 * i + 1];
      z[i] = z[2 * i + 1] && z[2 * i];
    }
  }
  return c[0];
}
```

B Double-Precision Multiplication

```
// Handle the special case of a zero, infinity, or NaN operand:

<ui117, ui8, bool, bool, bool>
specialCase(ui64 opa, ui64 opb, Class classa, Class classb,
            bool dn, bool fma, ui8 flags) {
  ui117 D = 0;
  ui64 zero = 0;
  zero[63] = opa[63] ^ opb[63];
  ui64 infinity = 0x7FF0000000000000 | zero;
  ui64 manMSB = 0x8000000000000;
  ui64 defNaN = 0x7FF8000000000000;
  bool piz = false;
```

```
  if (classa == SNAN) {
    D = dn ? defNaN : fma ? opa : opa | manMSB;
    flags[IOC] = 1; // invalid operand
  }
  else if (classb == SNAN) {
    D = dn ? defNaN : fma ? opb : opb | manMSB;
    flags[IOC] = 1; // invalid operand
  }
  else if (classa == QNAN) {
    D = dn ? defNaN : opa;
  }
  else if (classb == QNAN) {
    D = dn ? defNaN : opb;
  }
  else if (classa == INF && classb == ZERO ||
   classb == INF && classa == ZERO) {
    D = defNaN;
    piz = true;
    flags[IOC] = 1; // invalid operand
  }
  else if (classa == INF || classb == INF) {
    D = infinity;
  }
  else if (classa == ZERO || classb == ZERO) {
    D = zero;
  }
  if (fma) {
    D <<= 53;
  }
  bool inz = true, expGTinf = false;
  return <D, flags, piz, inz, expGTinf>;
}

// Compress the sum of 29 products to redundant form, using 27 3-2 compressors.

// Since the final sum is a 106-bit vector, the RTL limits every intermediate
// result to 106 bits.  The C model, however, in order to simplify the proof,
// does not.  This discrepancy does not affect the equivalence proof.

// For compressors receiving three inputs at the same time, t, the sum output
// emerges after 2 XOR delays (i.e. t+2) and the carry output after the
// equivalent of 1 XOR delay from a cgen cell (i.e. t+1).
// For compressors receiving two inputs at time t, and the third input at time
// t+1 (i.e. 1 XOR delay later), the sum and carry outputs both emerge at time
// t+2.  These timings are exploited to build reduction trees with minimum-depth
// logic.

<ui115, ui115> compress3to2(ui115 x, ui115 y, ui115 z) {
  ui115 sum = x ^ y ^ z;
  ui115 car = x & y | x & z | y & z;
  return <sum, car>;
}

<ui106, ui106> compress(ui57 pp[27], ui52 ia, ui53 ib) {

  // Time 0:
  ui59 t0fa0a = pp[0], t0fa0b = pp[1], t0fa0c = pp[2] << 2, t2pp0s, t1pp0c;
  <t2pp0s, t1pp0c> = compress3to2(t0fa0a, t0fa0b, t0fa0c);
  ui61 t0fa1a = pp[3], t0fa1b = pp[4] << 2, t0fa1c = pp[5] << 4, t2pp1s, t1pp1c;
  <t2pp1s, t1pp1c> = compress3to2(t0fa1a, t0fa1b, t0fa1c);
  ui61 t0fa2a = pp[6], t0fa2b = pp[7] << 2, t0fa2c = pp[8] << 4, t2pp2s, t1pp2c;
  <t2pp2s, t1pp2c> = compress3to2(t0fa2a, t0fa2b, t0fa2c);
  ui61 t0fa3a = pp[9], t0fa3b = pp[10] << 2, t0fa3c = pp[11] << 4, t2pp3s, t1pp3c;
  <t2pp3s,t1pp3c>  = compress3to2(t0fa3a, t0fa3b, t0fa3c);
  ui61 t0fa4a = pp[12], t0fa4b = pp[13] << 2, t0fa4c = pp[14] << 4, t2pp4s, t1pp4c;
```

```
<t2pp4s, t1pp4c> = compress3to2(t0fa4a, t0fa4b, t0fa4c);
ui61 t0fa5a = pp[15], t0fa5b = pp[16] << 2, t0fa5c = pp[17] << 4, t2pp5s, t1pp5c;
<t2pp5s, t1pp5c> = compress3to2(t0fa5a, t0fa5b, t0fa5c);
ui61 t0fa6a = pp[18], t0fa6b = pp[19] << 2, t0fa6c = pp[20] << 4, t2pp6s, t1pp6c;
<t2pp6s, t1pp6c> = compress3to2(t0fa6a, t0fa6b, t0fa6c);
ui61 t0fa7a = pp[21], t0fa7b = pp[22] << 2, t0fa7c = pp[23] << 4, t2pp7s, t1pp7c;
<t2pp7s, t1pp7c> = compress3to2(t0fa7a, t0fa7b, t0fa7c);
ui61 t0fa8a = pp[24], t0fa8b = pp[25] << 2, t0fa8c = pp[26] << 4, t2pp8s, t1pp8c;
<t2pp8s, t1pp8c> = compress3to2(t0fa8a, t0fa8b, t0fa8c);

// Time 1:
ui71 t1fa0a = t1pp0c, t1fa0b = t1pp1c << 4, t1fa0c = t1pp2c << 10, t3pp0s, t2pp0c;
<t3pp0s, t2pp0c> = compress3to2(t1fa0a, t1fa0b, t1fa0c);
ui73 t1fa1a = t1pp3c, t1fa1b = t1pp4c << 6, t1fa1c = t1pp5c << 12, t3pp1s, t2pp1c;
<t3pp1s, t2pp1c> = compress3to2(t1fa1a, t1fa1b, t1fa1c);
ui73 t1fa2a = t1pp6c, t1fa2b = t1pp7c << 6, t1fa2c = t1pp8c << 12, t3pp2s, t2pp2c;
<t3pp2s, t2pp2c> = compress3to2(t1fa2a, t1fa2b, t1fa2c);

// Time 2:
ui71 t2fa0a = t2pp0s, t2fa0b = t2pp1s << 4, t2fa0c = t2pp2s << 10, t4pp0s, t3pp0c;
<t4pp0s, t3pp0c> = compress3to2(t2fa0a, t2fa0b, t2fa0c);
ui73 t2fa1a = t2pp3s, t2fa1b = t2pp4s << 6, t2fa1c = t2pp5s << 12, t4pp1s, t3pp1c;
<t4pp1s, t3pp1c> = compress3to2(t2fa1a, t2fa1b, t2fa1c);
ui73 t2fa2a = t2pp6s, t2fa2b = t2pp7s << 6, t2fa2c = t2pp8s << 12, t4pp2s, t3pp2c;
<t4pp2s, t3pp2c> = compress3to2(t2fa2a, t2fa2b, t2fa2c);
ui107 t2fa3a = t2pp0c, t2fa3b = t2pp1c << 16, t2fa3c = t2pp2c << 34, t4pp3s, t3pp3c;
<t4pp3s, t3pp3c> = compress3to2(t2fa3a, t2fa3b, t2fa3c);

// Time 3:
ui107 t3fa0a = t3pp0s, t3fa0b = t3pp1s << 16, t3fa0c = t3pp2s << 34, t5pp0s, t4pp0c;
<t5pp0s, t4pp0c> = compress3to2(t3fa0a, t3fa0b, t3fa0c);
ui107 t3fa1a = t3pp0c, t3fa1b = t3pp1c << 16, t3fa1c = t3pp2c << 34, t5pp1s, t4pp1c;
<t5pp1s, t4pp1c> = compress3to2(t3fa1a, t3fa1b, t3fa1c);
ui107 t3fa2a = ia << 49, t3fa2b = ib << 49, t3fa2c = t3pp3c, t4pp4s, t4pp2c;
<t4pp4s, t4pp2c> = compress3to2(t3fa2a, t3fa2b, t3fa2c);

// Time 4:
ui109 t4fa0a = t4pp2c << 2, t4fa0b = t4pp1c, t4fa0c = t4pp0c, t6pp0s, t5pp0c;
<t6pp0s, t5pp0c> = compress3to2(t4fa0a, t4fa0b, t4fa0c);
ui110 t4fa1a = t4pp4s << 3, t4fa1b = t4pp0s, t4fa1c = t4pp1s << 16, t6pp1s, t5pp1c;
<t6pp1s, t5pp1c> = compress3to2(t4fa1a, t4fa1b, t4fa1c);

// Time 5:
ui111 t5fa0a = t5pp0s, t5fa0b = t5pp1s, t5fa0c =  t5pp0c << 2, t7pp0s, t6pp0c;
<t7pp0s, t6pp0c> = compress3to2(t5fa0a, t5fa0b, t5fa0c);
ui110 t5fa1a = t4pp2s << 33, t5fa1b = t4pp3s << 1, t5falc = t5pp1c, t6pp2s, t6pp1c;
<t6pp2s, t6pp1c> = compress3to2(t5fa1a, t5fa1b, t5fa1c);

// Time 6:
ui111 t6fa0a = t6pp0s << 2, t6fa0b = t6pp1s, t6fa0c = t6pp2s << 1, t8pp0s, t7pp0c;
<t8pp0s, t7pp0c> = compress3to2(t6fa0a, t6fa0b, t6fa0c);

// Time 7:
ui112 t7fa0a = t7pp0s, t7fa0b = t7pp0c,  t7fa0c = t6pp0c << 1, t9pp0s, t7pp1c;
<t9pp0s, t7pp1c> = compress3to2(t7fa0a, t7fa0b, t7fa0c);

// Time 8:
ui114 t8fa1a = t7pp1c << 2, t8fa1b = t6pp1c << 2, t8fa1c = t8pp0s, t9pp1s, t9pp0c;
<t9pp1s, t9pp0c> = compress3to2(t8fa1a, t8fa1b, t8fa1c);

// Time 9:
ui115 t9fa1a = t9pp0s << 1, t9fa1b = t9pp1s, t9fa1c = t9pp0c << 1, t11pp0s, t10pp0c;
<t11pp0s, t10pp0c> = compress3to2(t9fa1a, t9fa1b, t9fa1c);

ui115 ppa = t11pp0s;
```

```
  ui116 ppb = t10pp0c << 1;
  return <ppa, ppb>;
}

// Booth multiplier:

ui106 computeProduct(ui52 mana, ui52 manb, bool expaZero, bool expbZero) {
  ui57 pp[27]; // partial product array
  ui55 multiplier = manb;
  multiplier <<= 1;
  for (uint i = 0; i < 27; i++) {
    ui3 slice = multiplier[2 * i + 2:2 * i];
    bool sign = slice[2], signLast = slice[0];
    int enc = slice[0] + slice[1] - 2 * slice[2];
    ui53 mux;
    switch (enc) {
    case 0:
      mux = 0;
      break;
    case 1:
    case -1:
      mux = mana;
      break;
    case 2:
    case -2:
      mux = mana << 1;
    }
    if (sign) {
      mux = ~mux;
    }
    if (i == 0) {
      pp[i][52:0] = mux;
      pp[i][53] = sign;
      pp[i][54] = sign;
      pp[i][55] = !sign;
      pp[i][56] = 0;
    }
    else  {
      pp[i][0] = signLast;
      pp[i][1] = 0;
      pp[i][54:2] = mux;
      pp[i][55] = !sign;
      pp[i][56] = i < 26;
    }
  }
  ui52 ia = expaZero ? 0 : manb;
  ui53 ib = expbZero ? 0 : mana;
  ib[52] = !expaZero && !expbZero;
  ui106 ppa, ppb;
  <ppa, ppb> = compress(pp, ia, ib);
  return ppa + ppb;
}

// The design uses an internal exponent format: 12-bit signed integer with
// bias -1.  This function computes the internal representation of a biased
// 11-bit exponent, with 0 replaced by 1:

si12 expInt(ui11 expBiased) {
  ui12 expInt;
  expInt[11] = !expBiased[10];
  expInt[10] = !expBiased[10];
  expInt[9:1] = expBiased[9:1];
  expInt[0] = expBiased[0] || expBiased == 0;
  return expInt;
}
```

```
// Perform right shift if biased sum of exponents is 0 or negative:

<si12, bool, ui105, bool, bool, bool, bool>
rightShft(ui11 expa, ui11 expb, ui106 prod) {

  // Difference between 1 and biased sum of exponents:
  ui10 expDeficit = ~expa + ~expb + 1 + (expa != 0 && expb != 0);

  // If expDeficit >= 64, it may be replaced by 63 or 62:
  ui6 shift = expDeficit;
  if (expDeficit[9:6] != 0) {
    shift[5:1] = 31;
  }

  // Shifted product and fraction:
  ui107 prod0 = 0;
  prod0[106:1] = prod;
  ui106 prodShft = prod0 >> shift;
  ui105 frac105 = prodShft[104:0];
  si12 expShftInt = -0x400;
  bool expInc = prod[105] && (shift == 1);

  // Rounding bits:
  ui63 stkMaskFMA = 0;
  for (uint i = 0; i < shift; i++) {
    stkMaskFMA[i] = 1;
  }
  bool stkFMA = (prod & (stkMaskFMA >> 1)) != 0;
  ui107 stkMask = 0xFFFFFFFFFFFFF;
  stkMask[106:52] = stkMaskFMA[54:0];
  bool stk = (prod & stkMask[106:1]) != 0;
  ui55 grdMask = ~stkMask[106:52] & stkMask[105:51];
  bool grd = (grdMask & prod[105:51]) != 0;
  ui54 lsbMask = grdMask[53:0];
  bool lsb = (lsbMask & prod[105:52]) != 0;
  return <expShftInt, expInc, frac105, stkFMA, lsb, grd, stk>;
}

// Perform left shift if leading zero count is positive and biased sum
// of exponents is greater than 1:

<si12, bool, ui105, bool, bool, bool, bool>
leftShft(ui11 expa, ui11 expb, ui106 prod, ui6 clz) {

  // Internal representations of operand exponents:
  si12 expaInt = expInt(expa), expbInt = expInt(expb);

  // expProdInt - clz:
  si12 expDiffInt = expaInt + expbInt - clz + 1;
  si12 expProdM1Int = expaInt + expbInt;

  // Sign of biased sum of exponents:
  bool expDiffBiasedZero = expDiffInt == -0x400;
  bool expDiffBiasedNeg = expDiffInt < -0x400;
  bool expDiffBiasedPos = !expDiffBiasedZero && !expDiffBiasedNeg;

  // Shift amount:
  ui6 shift = expDiffBiasedZero ? clz - 1 :
              expDiffBiasedPos ? clz : expProdM1Int;

  // Shifted product and adjusted exponent:
  ui106 prodShft = prod << shift;
  si12 expShftInt = expDiffBiasedPos ? expDiffInt : -0x400;
```

```
  // Check for multiplication overflow:
  ui64 ovfMask = 0x8000000000000000 >> shift;
  bool mulOvf = (ovfMask & prod[105:42]) != 0;
  bool sub2Norm = ((ovfMask >> 1) & prod[104:42]) != 0;
  ui105 frac105 = prodShft[104:0];
  if (!mulOvf) {
    frac105 <<= 1;
  }

  // Condition for incrementing exponent:
  bool expInc = mulOvf || expDiffBiasedZero && sub2Norm;

  // Rounding bits:
  ui52 stkMask = 0xFFFFFFFFFFFFF >> shift;
  bool stk = mulOvf ? (stkMask & prod) != 0 : ((stkMask >> 1) & prod) != 0;
  ui53 grdMask = ovfMask[63:11];
  bool grd = mulOvf ? (grdMask & prod) != 0 : ((grdMask >> 1) & prod) != 0;
  ui54 lsbMask = ovfMask[63:10];
  bool lsb = mulOvf ? (lsbMask & prod) != 0 : ((lsbMask >> 1) & prod) != 0;
  return <expShftInt, expInc, frac105, 0, lsb, grd, stk>;
}

// Inputs of fmul64:
//   opa[63:0], opb[63:0]: sign 63, exponent 62:52, mantissa 51:0
//   fz: force denormals to 0
//   dn: replace NaN operand with default
//   mode[1:0]: encoding of rounding mode
//   fma: boolean indication of FMA rather than FMUL

// Outputs of fmul64:
//   D[116:0]: For FMUL, data result is D[63:0];
//             for FMA, sign 116, exponent 115:105, mantissa 104:0
//   flags[7:0]: exception flags
//   piz: product of infinity and zero (valid for FMA only)
//   inz: result is infinity, NaN, or zero (valid for FMA only)
//   expOvfl: implicit exponent bit 11 (valid for FMA when inz = 0)

<ui117, ui8, bool, bool, bool>
fmul64(ui64 opa, ui64 opb, bool fz, bool dn, ui2 rmode, bool fma) {

  // Analyze operands and process special cases:
  bool signa, signb;    // operand signs
  ui11 expa, expb;      // operand exponents
  ui52 mana, manb;      // operand mantissas
  Class classa, classb; // operand classes
  ui8 flags = 0;        // exception flags
  <signa, expa, mana, classa> = analyze(opa, DP, fz, flags);
  <signb, expb, manb, classb> = analyze(opb, DP, fz, flags);

  // Detect early exit:
  if (classa == ZERO || classa == INF || classa == SNAN || classa == QNAN ||
      classb == ZERO || classb == INF || classb == SNAN || classb == QNAN) {
    return specialCase(opa, opb, classa, classb, dn, fma, flags);
  }
  else  {

    // Leading zero count:
    ui6 clz = 0;
    if (expa == 0) {
      clz |= CLZ53(mana);
    }
    if (expb == 0) {
      clz |= CLZ53(manb);
    }
```

```
// Product of significands:
ui106 prod = computeProduct(mana, manb, expa == 0, expb == 0);

// Internal representations of operand exponents and their sum:
si12 expaInt = expInt(expa), expbInt = expInt(expb);
si12 expProdInt = expaInt + expbInt + 1;

// Biased sum of exponents is 0, negative:
bool expBiasedZero = expProdInt == -0x400;
bool expBiasedNeg = expProdInt < -0x400;

// If biased sum is 0 or negative, a right  shift is required.
// Otherwise, a left shift (possibly 0) is performed.
// Iin both cases, we compute the following quantities:
si12 expShftInt;     // expShftInt + expInc is internal representation
bool expInc;         //  of exponent of shifted product
ui105 frac105;       // fraction to be returned for FMA
bool stkFMA;         // sticky bit for FMA
bool lsb, grd, stk;  // lsb, guard, and sticky bits for FMUL

if (expBiasedZero || expBiasedNeg) {
  <expShftInt, expInc, frac105, stkFMA> = rightShft(expa, expb, prod);
}
else  {
  <expShftInt, expInc, frac105, stkFMA> = leftShft(expa, expb, prod, clz);
}

// Important values of (pre-increment) exponent:
bool expZero = expShftInt == -0x400;
bool expMax = expShftInt == 0x3FE;
bool expInf = expShftInt == 0x3FF;
bool expGTinf = expShftInt >= 0x400;

// Convert exponent to biased form:
ui11 exp11 = expShftInt;
exp11[10] = !exp11[10];

// Sign of product:
bool sign = signa ^ signb;

if (fma) { // FMA case
  ui117 D;
  D[116] = sign;
  if (expInc && !expInf) {
    D[115:105] = exp11 + 1;
  }
  else  {
    D[115:105] = exp11;
  }
  D[104:0] = frac105;
  flags[IXC] = stkFMA;
  bool piz = false, inz = false;
  bool expOvfl = expGTinf || expInf && expInc;
  return <D, flags, piz, inz, expOvfl>;
}

else  { // FMUL case
  ui64 D = 0;
  D[63] = sign;
  bool rndUp = rmode == rmodeNear && grd && (lsb || stk) ||
           rmode == rmodeUP && !sign && (grd || stk) ||
           rmode == rmodeDN && sign && (grd || stk);
  ui52 fracUnrnd = frac105[104:53];
  ui53 fracP1 = fracUnrnd + 1;
  ui52 fracRnd = rndUp ? fracP1[51:0] : fracUnrnd;
```

```
      bool expRndInc = rndUp && fracP1[52];
      ui11 expRnd = expInc || expRndInc ? exp11 + 1 : exp11;
      bool underflow = expZero && !expInc;
      bool overflow = expGTinf || expInf || expMax && (expInc || expRndInc);
      if (overflow) {
        flags[IXC] = 1;
        flags[OFC] = 1;
        if (rmode == rmodeUP && sign || rmode == rmodeDN && !sign ||
         rmode == rmodeZero) {
          D[62:0] = 0x7FEFFFFFFFFFFFFF;
        }
        else {
          D[62:0] = 0x7FF0000000000000;
        }
      }
      else if (underflow) {
        if (fz) {
          flags[UFC] = 1;
        }
        else {
          if (grd || stk) {
            flags[UFC] = 1;
            flags[IXC] = 1;
          }
          D[51:0] = fracRnd;
          D[62:52] = expRnd;
        }
      }
      else {
        if (grd || stk) {
          flags[IXC] = 1;
        }
        D[51:0] = fracRnd;
        D[62:52] = expRnd;
      }
      return <D, flags, false, false, false>;
    }
  }
}
```

C Double-Precision Addition with FMA

```
// Rounding direction:

enum RndDir {rndNear, rndZero, rndInf};

RndDir computeRndDir(ui2 rmode, bool sign) {
  if (rmode == rmodeNear) {
    return rndNear;
  }
  else if (rmode == rmodeZero || rmode == rmodeUP && sign || rmode == rmodeDN && !sign) {
    return rndZero;
  }
  else {
    return rndInf;
  }
}

// Components of 117-bit operand:

bool sign(ui117 op) {
  return op[116];
```

```
}

ui11 expnt(ui117 op) {
  return op[115:105];
}

ui105 frac(ui117 op) {
  return op[104:0];
}

// Apply FZ to denormal operands:

<ui117, ui8> checkDenorm(ui117 op, ui8 flags, bool fz) {
  if (fz && expnt(op) == 0 && frac(op) != 0) {
    op[104:0] = 0;
    flags[IDC] = 1;
  }
  return <op, flags>;
}

// Identify special case (NaN or infinity operand, invalid op, or zero sum) and
// if detected, return data result and updated flags:

<bool, ui64, ui8>
checkSpecial(ui117 opa, ui117 opp, bool fz, bool dn, ui2 rmode,
bool oppLong,
             bool mulOvfl, bool piz, bool mulStk, ui8 flags) {

  bool signa = sign(opa), signp = sign(opp);
  ui11 expa = expnt(opa), expp = expnt(opp);
  ui105 fraca = frac(opa), fracp = frac(opp);

  bool opaZero = (expa == 0) && (fraca == 0);
  bool opaInf = (expa == 0x7FF) && (fraca == 0);
  bool opaNan = (expa == 0x7FF) && (fraca != 0);
  bool opaQnan = opaNan && fraca[104];
  bool opaSnan = opaNan && !fraca[104];
  bool oppZero = (expp == 0) && (fracp == 0) && !mulOvfl && !mulStk;
  bool oppInf = (expp == 0x7FF) && (fracp == 0) && !oppLong;
  bool oppNan = (expp == 0x7FF) && (fracp != 0) && !oppLong;
  bool oppQnan = oppNan && fracp[104];
  bool oppSnan = oppNan && !fracp[104];
  ui64 DefNan = 0x7FF8000000000000;

  bool isSpecial = false;
  ui64 D = 0;

  if (opaSnan) {
    isSpecial = true;
    D = dn ? DefNan : opa[116:53] | 0x0008000000000000;
    flags[IOC] = 1; // invalid operand
  }
  else if (piz) {
    isSpecial = true;
    D = DefNan;
    // IOC is already set in mulExcps, so needn't be set here
  }
  else if (oppSnan) {
    isSpecial = true;
    D = dn ? DefNan : opp[116:53] | 0x0008000000000000;
    flags[IOC] = 1; // invalid operand
  }
  else if (opaQnan) {
    isSpecial = true;
    D = dn ? DefNan : opa[116:53];
  }
  else if (oppQnan) {
```

```
    isSpecial = true;
    D = dn ? DefNan : opp[116:53];
  }
  else if (opaInf) {
    isSpecial = true;
    if (oppInf && signa != signp) {
      D = DefNan;
      flags[IOC] = 1; // invalid operand
    }
    else  {
      D = opa[116:53];
    }
  }
  else if (oppInf) {
    isSpecial = true;
    D = opp[116:53];
  }
  else if (opaZero && oppZero && signa == signp) {
    isSpecial = true;
    D[63] = signa;
  }
  else if (expa == expp && fraca == fracp && !mulOvfl && !mulStk && signa != signp) {
    isSpecial = true;
    if (rmode == rmodeDN) {
      D[63] = 1;
    }
  }
  return <isSpecial, D, flags>;
}

// Determine near or far path:

bool isFar(ui11 expa, ui11 expp, bool usa) {
  ui12 expaP1 = expa + 1;
  ui12 exppP1 = expp + 1;
  bool isNear = usa && (expa == expp || expa == exppP1 || expp == expaP1);
  return !isNear;
}

// Compute sum and return absolute value, sticky bit, and sign:

<ui108, bool, bool> add(ui117 opa, ui117 opp, bool far, bool usa, bool mulStk) {

  bool signa = sign(opa), signp = sign(opp);
  ui11 expa = expnt(opa), expp = expnt(opp);
  ui105 fraca = frac(opa), fracp = frac(opp), fracl, fracs;
  bool oppGEopa = expp > expa || expp == expa && fracp  >= fraca;

  // Construct significands, padding with a zero at the top to allow for overflow
  // in the far case, and a zero at the bottom to allow for a 1-bit right shift on
  // the near path:
  ui108 siga = 0;
  siga[106] = expa != 0;
  siga[105:1] = fraca;
  ui108 sigp = 0;
  sigp[106] = expp != 0;
  sigp[105:1] = fracp;

  // In the case far && !usa, the leading 1 of the sum or difference is at bit 107
  // or 106.  The LZA is designed so that the same is true of the shifted sum in
  // the near case.  In order to for this hold in the case far && usa, we perform
  // a 1-bit left shift:
  ui108 sigaPrime = siga, sigpPrime = sigp;
  if (far && usa) {
    sigaPrime <<= 1;
    sigpPrime <<= 1;
  }
```

```
 // Compare the operands and determine the exponent difference for the right shift
 // of the smaller one.  For this purpose, the exponent of a subnormal operand is
 // taken to be 1 rather than 0:
  bool signl; // sign of the result
  ui108 sigl, sigs; // significands of larger and smaller operands
  ui12 expDiff;
  if (oppGEopa) {
    signl = signp;
    sigl = sigpPrime;
    sigs = sigaPrime;
    if (expa == 0 && expp != 0) {
      expDiff = expp - expa - 1;
    }
    else  {
      expDiff = expp - expa;
    }
  }
  else  {
    signl = signa;
    sigl = sigaPrime;
    sigs = sigpPrime;
    if (expp == 0 && expa != 0) {
      expDiff = expa - expp - 1;
    }
    else  {
      expDiff = expa - expp;
    }
  }

  // If the right shift exceeds the significand width, its value is uninteresting.
  // Therefore, we can collapse the 8 bits expDiff[11:4] to 3 bits as follows:
  ui7 rshift = expDiff[6:0];
  if (expDiff[11:7] != 0) {
    rshift |= 0x70;
  }

  ui108 sigShft = sigs >> rshift;
  bool shiftOut = (sigShft << rshift) != sigs;

  // Compute the sum or difference and the sticky bit.  In the case of subtraction,
  // if either (a) sigs = sigp and mulStk = 1 or (b) a nonzero value has been shifted
  // out, then the computed difference is an overestimate rather then an underestimate.
  // In this event, we decrement the difference by eliminating the carry-in:
  bool cin = usa && !(mulStk && !oppGEopa || far && shiftOut);
  ui108 ops = usa ? ~sigShft : sigShft;
  ui108 sum = sigl + ops + cin;
  bool stk = mulStk || far && shiftOut;

  return <sum, stk, signl>;
}

// Count leading zeroes of a + b, where a and b are 128-bit vectors,
// under these assumptions:
//   (1) the 128-bit sum is not 0;
//   (2) the addition produces a carry-out
// The result may be an overestimate by 1:

ui7 LZA128(ui128 a, ui128 b) {

  // Let n be index of the lsb of the maximal prefix of the form P*GK*
  // (where P is propagate, G is generate, K is kill). Then n > 0 and
  // the index of the leading 1 of the sum is either n or n-1.

  // Construct a vector w that has its leading 1 at index n:
  ui128 p = a ^ b;
  ui128 k = ~a & ~b;
```

```
  // w[i] = ~z[i], where
  //   z[i] = (p[i] & p[i-1]) | (p[i] & g[i-1]) | (g[i] & k[i-1]) | (k[i] & k[i-1])
  //        = (p[i] & (p[i-1] | g[i-1])) | ((g[i] | k[i]) & k[i-1])
  //        = (p[i] & ~k[i-1]) | (~p[i] & k[i-1])
  //        = p[i] ^ k[i-1]

  ui128 w = ~(p ^ (k << 1));

  // Now the number of leading zeroes of w is either equal to the number of
  // leading zeroes of the sum or 1 less, so we pad it with an extra leading zero:
  return CLZ128(w >> 1);
}

// Compute leading zero count of the difference in the near case:

ui7 computeLZA(ui117 opa, ui117 opp) {
  ui128 in1LZA = 0, in2LZA = 0;
  ui11 expa = expnt(opa), expp = expnt(opp);
  ui105 fraca = frac(opa), fracp = frac(opp), fracl, fracs;
  bool oppGEopa = expp > expa || expp == expa && fracp >= fraca;
  if (oppGEopa) {
    fracl = fracp;
    fracs = fraca;
  }
  else  {
    fracl = fraca;
    fracs = fracp;
  }
  in1LZA[127] = 1;
  in1LZA[126:22] = fracl;
  if (expp[0] == expa[0]) {
    in2LZA = (1 << 22) - 1;
    in2LZA[126:22] = ~fracs;
  }
  else  {
    in2LZA = (1 << 21) - 1;
    in2LZA[125:21] = ~fracs;
    in2LZA[127] = 1;
  }
  return LZA128(in1LZA, in2LZA);
}

// Compute left shift and adjusted exponent:

<ui7, ui12> computeLshift(ui117 opa, ui117 opp, bool far, bool usa) {
  ui11 expa = expnt(opa), expp = expnt(opp);
  ui12 expl = expa >= expp ? expa : expp;
  ui7 lshift;   // left shift
  ui12 expShft; // adjusted exponent
  ui7 lza = computeLZA(opa, opp);
  if (far) {
    lshift = 0;
    expShft = usa ? expl - 1 : expl;
  }
  else if (lza < expl) {
    lshift = lza;
    expShft = expl - lza;
  }
  else  {
    lshift = expl == 0 ? 0 : expl - 1;
    expShft = 0;
  }
  return <lshift, expShft>;
}
```

```
// The rounding increments and inexact bits for the overflow and non-overflow cases
// are computed during the left shift.  This is done by applying lsb, guard, and
// sticky masks to the unshifted sum.  Thus, the masks must be right-shifted by the
// left shift amount.  This may be done as soon as the shift amount is known:

<bool, bool, bool, bool> rndInfo(ui108 sum, bool stk, ui7 lshift, RndDir rndDir) {

  // lsb, guard, and sticky masks:
  ui56 lOvflMask = 0x80000000000000 >> lshift;
  ui55 gOvflMask = lOvflMask >> 1;
  ui54 sOvflMask = 0x3FFFFFFFFFFFFF >> lshift;
  ui55 lNormMask = lOvflMask >> 1;
  ui54 gNormMask = lOvflMask >> 2;
  ui53 sNormMask = sOvflMask >> 1;

  // lsb, guard, and sticky bits:
  bool lOvfl = (sum & lOvflMask) != 0;
  bool gOvfl = (sum & gOvflMask) != 0;
  bool sOvfl = (sum & sOvflMask) != 0 || stk;
  bool lNorm = (sum & lNormMask) != 0;
  bool gNorm = (sum & gNormMask) != 0;
  bool sNorm = (sum & sNormMask) != 0 || stk;

  // rounding increments;
  bool incOvfl = (rndDir == rndNear) && gOvfl && (lOvfl || sOvfl) ||
                 (rndDir == rndInf) && (gOvfl || sOvfl);
  bool incNorm = (rndDir == rndNear) && gNorm && (lNorm || sNorm) ||
                 (rndDir == rndInf) && (gNorm || sNorm);

  // inexact bits:
  bool inxOvfl = gOvfl || sOvfl;
  bool inxNorm = gNorm || sNorm;

  return <incOvfl, incNorm, inxOvfl, inxNorm>;
}

// Inputs of fadd64:
//   opa[63:0]: sign 63, exponent 62:52, mantissa 51:0
//   opp[116:0]: sign 116, exponent 115:105, mantissa 104:0
//   fz, dn, rmode: FPSCR components
//   fma: fused mul-add
//   inz: multiplier output is infinity, NaN, or zero
//   piz: multiplier computes inf * 0 and returns DefNan
//   expOvfl: bit 11 of opp exponent from multiplier
//   mulExcps[7:0]: exception flags from multiplier

// Outputs of fadd64:
//   D[63:0]: data result
//   flags[7:0]: exception flags

<ui64, ui8>
fadd64(ui64 opa, ui117 opp, bool fz, bool dn, ui2 rmode, bool fma, bool inz,
       bool piz, bool expOvfl, ui8 mulExcps) {

  ui64 D; // data result
  ui8 flags = 0; // initialize flags

  // An fma with a NaN, infinity, or zero from the multiplier is treated as
  // an ordinary add:
  bool oppLong = fma && !inz;

  // expOvfl is qualified by oppLong:
  bool mulOvfl = oppLong && expOvfl;

  // piz is qualified by fma:
  piz = fma && piz;
```

```
// In fma case, mulExcps[IXC] is sticky bit from multiplier:
bool mulStk = mulExcps[IXC] && oppLong;

// In fma case, copy flags from multiplier, ignoring mulExcps[IXC] when
// it is sticky bit:
if (fma) {
  flags = mulExcps;
  flags[IXC] = flags[IXC] && !oppLong;
}

// opa extended to 117 bits:
ui117 opax = 0;
opax[116:53] = opa;

// Apply FZ to denormal operands:
ui117 opaz, oppz;
<opaz, flags> = checkDenorm(opax, flags, fz);
if (!fma) {
  <oppz, flags> = checkDenorm(opp, flags, fz);
}
else  {
  oppz = opp;
}

// NaN or infinity operand, invalid op, or zero sum:
bool isSpecial;
<isSpecial, D, flags> =
  checkSpecial(opaz, oppz, fz, dn rmode, oppLong, mulOvfl, piz, mulStk, flags);
if (isSpecial) {
  return <D, flags>;
}

// Nonzero sum:
else  {
  // Unlike signs:
  bool usa = sign(opaz) != sign(oppz);

  // Far path (unlike signs and exponents within 1):
  bool far = isFar(expnt(opaz), expnt(oppz), usa);

  // Perform right shift and compute sum:
  ui108 sum;
  bool stk, signl;
  <sum, stk, signl> = add(opaz, oppz, far, usa, mulStk);

  // Compute left shift and adjusted exponent (concurrent with addition):
  ui7 lshift;
  ui12 expShft;
  <lshift, expShft> = computeLshift(opaz, oppz, far, usa);

  // Perform the left shift:
  ui108 sumShft = sum << lshift;

  // Sign of result:
  bool signOut = mulOvfl ? sign(opp) : signl;

  // Rounding direction:
  RndDir rndDir = computeRndDir(rmode, signOut);

  // Compute rounding increments and inexact bits while shifting is performed:
  bool incOvfl, incNorm, inxOvfl, inxNorm;
  <incOvfl, incNorm, inxOvfl, inxNorm> = rndInfo(sum, stk, lshift, rndDir);

  // Rounding may be done as soon as the shifted sum is available:
  ui54 sumUnrnd = sumShft[107:54]; // unrounded sum, with 2 integer bits
  ui54 sumNorm = sumUnrnd + incNorm; // rounded sum, assuming no overflow
  ui54 sumOvfl = sumUnrnd[53:1] + incOvfl; // rounded sum, assuming overflow
```

```
    // Case analysis:
    bool tiny = !sumUnrnd[53] && !sumUnrnd[52]; // unrounded sum is subnormal
    bool ovfl = sumNorm[53]; // overflow
    bool ovfl2 = (sumUnrnd[53:1] == ((1 << 53) - 1)) && incOvfl; // double overflow
    bool infOrMax = expShft == 0x7FE && (ovfl || ovfl2) || expShft == 0x7FD && ovfl2 ||
                    expShft == 0x7FF && oppLong || mulOvfl; // rounded sum is supernormal

    // Computation of final result and exception flags:
    ui11 expOut;
    ui52 fracOut;
    if (infOrMax) { // supernormal rounded result
      if (rndDir == rndZero) { // return largest normal
        expOut = 0x7FE;
        fracOut = 0xFFFFFFFFFFFFF;
      }
      else  { // return infinity
        expOut = 0x7FF;
        fracOut = 0;
      }
      flags[OFC] = 1; // overflow
      flags[IXC] = 1; // inexact
    }
    else if (tiny) { // subnormal unrounded result
      if (fz) { // flush to zero
        expOut = 0;
        fracOut = 0;
        flags[UFC] = 1; // underflow but not inexact
      }
      else if (sumNorm[52]) { // rounded up to normal
        expOut = 1;
        fracOut = 0;
        flags[UFC] = 1; // underflow
        flags[IXC] = 1; // inexact
      }
      else  {// rounded result is subnormal
        expOut = expShft;  // expOut = 0
        fracOut = sumNorm[51:0];
        if (inxNorm) {
          flags[UFC] = 1; // underflow
          flags[IXC] = 1; // inexact
        }
      }
    }
    else if (ovfl2) { // double overflow
      expOut = expShft + 2;
      fracOut = 0;
      flags[IXC] = flags[IXC] || inxOvfl; // inexact
    }
    else if (ovfl) { // overflow or double overflow of subnormal
      expOut = expShft == 0 ? 2 : expShft + 1;
      fracOut = sumOvfl[51:0];
      flags[IXC] = flags[IXC] || inxOvfl; // inexact
    }
    else  { // overflow of subnormal
      expOut = expShft == 0 && sumNorm[52] ? 1 : expShft;
      fracOut = sumNorm[51:0];
      flags[IXC] = flags[IXC] || inxNorm; // inexact
    }
    D[63] = signOut;
    D[62:52] = expOut;
    D[51:0] = fracOut;

    return <D, flags>;
  }
}
```

D Multi-Precision Radix-4 Division

```
// Compute Q, incremented Q, and sticky bit (shared with fsqrt64):

<ui53, ui53, bool> computeQ(ui54 QP, ui54 QN, ui59 RP, ui59 RN, ui2 fmt, bool isSqrt) {

  // Sign of remainder:
  ui59 rem = RP + ~RN + 1;
  bool remSign = rem[58];
  bool remZero = (RP ^ RN) == 0;

  // If the remainder is negative, then the quotient must be decremented.  This is
  // achieved by eliminating the carry-in bit:
  bool cin = !remSign;

  // If the sum is to be rounded up, then a rounding increment is added.  Note that
  // the position of the increment is the lsb of the result. For fdiv, this is bit 1
  // for SP and bit 2 for DP and HP; for fsqrt, it is the opposite:
  bool lsbIs2 = isSqrt == (fmt == SP);
  ui3 inc = lsbIs2 ? 4 : 2;

  // RTL computes 4 sums in parallel with the rounding increment:
  //   Q0     cin = 0, inc = 0
  //   Q0inc  cin = 0, inc > 0
  //   Q1     cin = 1, inc = 0
  //   Q1inc  cin = 1, inc > 0

  // Two adders are used to compute Q0 and Q1inc; the other sums are derived from these.
  // The simplest sum is Q0:

  ui54 Q0 = QP + ~QN;

  // In order to compute Q1inc, inc is added in via a 3-2 compressor.
  ui54 QN1inc = QP ^ ~QN ^ inc;
  ui54 QP1inc = (QP & ~QN | (QP | ~QN) & inc) << 1;
  ui54 Q1inc = QP1inc + QN1inc + 1;

  // For the other two sums, first we compute the bottom 3 bits:
  ui3 Q1Low = QP[2:0] + ~QN[2:0] + 1;
  ui3 Q0incLow = QP1inc[2:0] + QN1inc[2:0];
  ui54 Q1 = Q1Low;
  ui54 Q0inc = Q0incLow;

  // The upper bits are just copied (note the difference between fdiv and fsqrt):
  if (Q1 == 0) {
    Q1[53:3] = Q1inc[53:3];
  }
  else  {
    Q1[53:3] = Q0[53:3];
  }
  if (Q0inc <= 1 || Q0inc <= 3 && lsbIs2) {
    Q0inc[53:3] = Q1inc[53:3];
  }
  else  {
    Q0inc[53:3] = Q0[53:3];
  }

  // When cin is finally available, the following selections are made:
  ui54 Q01 = cin ? Q1 : Q0;
  ui54 Q01inc = cin ? Q1inc : Q0inc;

  // Discard the extra bit if present:
  ui53 Qtrunc = lsbIs2 ? Q01 >> 1 : Q01;
  ui53 Qinc = lsbIs2 ? Q01inc >> 1 : Q01inc;
  return <Qtrunc, Qinc, !remZero>;
}

// Right-shift a 64-bit vector:
```

```
<ui64, bool> rShft64(ui64 x, ui6 s) {
  ui64 xs = x >> s;
  bool stk = x != (xs << s);
  return <xs, stk>;
}

// Compute rounded result for both normal and denormal cases (shared with fsqrt64):

<ui53, bool, ui53, bool>
rounder(ui53 Qtrunc, ui53 Qinc, bool stk, bool sign, si13 expQ, ui2 rmode, ui2 fmt) {

  // Rounding decision for normal case:
  bool lsb = Qtrunc[1], grd = Qtrunc[0];
  ui53 Qrnd;
  if ((rmode == rmodeNear) && grd && (lsb || stk) ||
      (rmode == rmodeUP) && !sign && (grd || stk) ||
      (rmode == rmodeDN) && sign && (grd || stk)) {
    Qrnd = Qinc[53:1];
  }
  else  {
    Qrnd = Qtrunc[53:1];
  }
  bool inx = grd || stk;

  // Right-shifted quotient and rounding decision for subnormal case:
  ui64 QDen = 0; // Insert integer bit
  switch (fmt) {
  case DP:
    QDen[53] = 1;
    QDen[52:0] = Qtrunc[52:0];
    break;
  case SP:
    QDen[24] = 1;
    QDen[23:0] = Qtrunc[23:0];
    break;
  case HP:
    QDen[11] = 1;
    QDen[10:0] = Qtrunc[10:0];
  }
  ui12 shft12 = 1 - expQ; // shift is at most 63
  ui6 shft = shft12 >= 64 ? 63 : shft12;
  bool lsbDen, grdDen, stkDen;
  ui64 Qshft;
  <Qshft, stkDen> = rShft64(QDen, shft);
  lsbDen = Qshft[1];
  grdDen = Qshft[0];
  stkDen = stkDen || stk;
  ui54 QrndDen;
  if ((rmode == rmodeNear) && grdDen && (lsbDen || stkDen) ||
      (rmode == rmodeUP) && !sign && (grdDen || stkDen) ||
      (rmode == rmodeDN) && sign && (grdDen || stkDen)) {
    QrndDen = Qshft[53:1] + 1;
  }
  else  {
    QrndDen = Qshft[53:1];
  }
  bool inxDen = grdDen || stkDen;
  return <Qrnd, inx, QrndDen, inxDen>;
}

// Final result (shared with fsqrt64):

<ui64, ui8>
final(ui53 Qrnd, bool inx, ui53 QrndDen, bool inxDen, bool sign,
      si13 expQ, ui2 rmode, bool fz, ui2 fmt, ui8 flags) {

  // Selection of infinity or max normal for overflow case:
  bool selMaxNorm = rmode == rmodeDN && !sign ||
                    rmode == rmodeUP && sign ||
                    rmode == rmodeZero;
```

```
ui64 D = 0;  // data result
switch (fmt) {
case DP:
  D[63] = sign;
  if (expQ >= 0x7FF) { // overflow
    if (selMaxNorm) {
      D[62:52] = 0x7FE;
      D[51:0] = 0xFFFFFFFFFFFFF;
    }
    else  {
      D[62:52] = 0x7FF;
      D[51:0] = 0;
    }
    flags[OFC] = 1; // overflow
    flags[IXC] = 1; // inexact
  }
  else if (expQ <= 0) { // subnormal
    if (fz) {
      flags[UFC] = 1; // underflow but not inexact
    }
    else  {
      ui11 exp = QrndDen[52];
      D[62:52] = exp;
      D[51:0] = QrndDen[51:0];
      flags[IXC] = flags[IXC] || inxDen;
      flags[UFC] = flags[UFC] || inxDen;
    }
  }
  else  { // normal
    D[62:52] = expQ;
    D[51:0] = Qrnd[51:0];
    flags[IXC] = flags[IXC] || inx;
  }
  break;
case SP:
  D[31] = sign;
  if (expQ >= 0xFF) { // overflow
    if (selMaxNorm) {
      D[30:23] = 0xFE;
      D[22:0] = 0x7FFFFF;
    }
    else  {
      D[30:23] = 0xFF;
      D[22:0] = 0;
    }
    flags[OFC] = 1; // overflow
    flags[IXC] = 1; // inexact
  }
  else if (expQ <= 0) { // subnormal
    if (fz) {
      flags[UFC] = 1; // underflow but not inexact
    }
    else  {
      ui8 exp = QrndDen[23];
      D[30:23] = exp;
      D[22:0] = QrndDen[22:0];
      flags[IXC] = flags[IXC] || inxDen;
      flags[UFC] = flags[UFC] || inxDen;
    }
  }
  else  { // normal
    D[30:23] = expQ;
    D[22:0] = Qrnd[22:0];
    flags[IXC] = flags[IXC] || inx;
  }
  break;
case HP:
  D[15] = sign;
  if (expQ >= 0x1F) { // overflow
    if (selMaxNorm) {
```

```
        D[14:10] = 0x1E;
        D[9:0] = 0x3FF;
      }
      else  {
        D[14:10] = 0x1F;
        D[9:0] = 0;
      }
      flags[OFC] = 1; // overflow
      flags[IXC] = 1; // inexact
    }
    else if (expQ <= 0) { // subnormal
      if (fz) {
        flags[UFC] = 1; // underflow but not inexact
      }
      else  {
        ui5 exp = QrndDen[10];
        D[14:10] = exp;
        D[9:0] = QrndDen[9:0];
        flags[IXC] = flags[IXC] || inxDen;
        flags[UFC] = flags[UFC] || inxDen;
      }
    }
    else  {
      D[14:10] = expQ;
      D[9:0] = Qrnd[9:0];
      flags[IXC] = flags[IXC] || inx;
    }
    break;
  }
  return <D, flags>;
}

// Zero, infinity, or NaN:

<ui64, ui8>
specialCase(bool sign, ui64 opa, ui64 opb, Class classa, Class classb,
    ui2 fmt, bool dn, ui8 flags) {
  bool isSpecial = false;
  ui64 D = 0;
  ui64 aNan, bNan, manMSB, infinity, defNaN, zero = 0;
  switch (fmt) {
  case DP:
    aNan = opa[63:0];
    bNan = opb[63:0];
    zero[63] = sign;
    infinity = 0x7FF0000000000000;
    manMSB = 0x8000000000000;
    break;
  case SP:
    aNan = opa[31:0];
    bNan = opb[31:0];
    zero[31] = sign;
    infinity = 0x7F800000;
    manMSB = 0x400000;
    break;
  case HP:
    aNan = opa[15:0];
    bNan = opb[15:0];
    zero[15] = sign;
    infinity = 0x7C00;
    manMSB = 0x200;
    break;
  }
  defNaN = infinity | manMSB;
  if (classa == SNAN) {
    D = dn ? defNaN : aNan | manMSB;
    flags[IOC] = 1; // invalid operand
  }
  else if (classb == SNAN) {
    D = dn ? defNaN : bNan | manMSB;
```

```
    flags[IOC] = 1; // invalid operand
  }
  else if (classa == QNAN) {
    D = dn ? defNaN : aNan;
  }
  else if (classb == QNAN) {
    D = dn ? defNaN : bNan;
  }
  else if (classa == INF) {
    if (classb == INF) {
      D = defNaN;
      flags[IOC] = 1; // invalid operand
    }
    else  {
      D = infinity | zero;
    }
  }
  else if (classb == INF) {
    D = zero;
  }
  else if (classa == ZERO) {
    if (classb == ZERO) {
      D = defNaN;
      flags[IOC] = 1; // invalid operand
    }
    else  {
      D = zero;
    }
  }
  else if (classb == ZERO) {
    D = infinity | zero;
    flags[DZC] = 1; // division by 0
  }
  return <D, flags>;
}

// Normalize denormal operands and compute exponent difference:

<ui53, ui53, si13> normalize(ui11 expa, ui11 expb, ui52 mana, ui52 manb, ui2 fmt) {
  ui53 siga = 0, sigb = 0;
  uint bias;
  switch (fmt) {
  case DP:
    siga = mana;
    sigb = manb;
    bias = 0x3FF;
    break;
  case SP:
    siga[51:29] = mana;
    sigb[51:29] = manb;
    bias = 0x7F;
    break;
  case HP:
    siga[51:42] = mana;
    sigb[51:42] = manb;
    bias = 0xF;
  }
  si13 expaShft, expbShft;
  if (expa == 0) {
    ui6 clz = CLZ53(siga);
    siga <<= clz;
    expaShft = 1 - clz;
  }
  else  {
    siga[52] = 1;
    expaShft = expa;
  }
  if (expb == 0) {
    ui6 clz = CLZ53(sigb);
    sigb <<= clz;
```

```
    expbShft = 1 - clz;
  }
  else  {
    sigb[52] = 1;
    expbShft = expb;
  }
  si13 expDiff = expaShft - expbShft + bias;
  return <siga, sigb, expDiff>;
}

// Prescale the divisor d and the dividend x = 4R0.
// Use the redundant form of x to compute q1.
// Convert d and x to non-redundant form.
// Shift x 1 bit if necessary to ensure that the quotient is in [1, 2) and
// decrement the quotient exponent accordingly.
// Return d along with q1*d and x, which are the sum and carry vectors of R1,
// and the quotient exponent.

<ui57, ui59, ui59, si13, int> prescale(ui56 siga, ui56 sigb, si13 expDiff) {
  ui56 div1, div2, div3, divSum, divCar;
  if (!sigb[51] && (sigb[50] || !sigb[49])) {
    div1 = sigb << 2;
  }
  else if (!sigb[50] && (sigb[51] || sigb[49])) {
    div1 = sigb << 1;
  }
  else  {
    div1 = 0;
  }
  if (!sigb[51] && !sigb[50]) {
    div2 = sigb << 2;
  }
  else if ((sigb[51] || sigb[50]) && !sigb[49] || sigb[51] && sigb[50]) {
    div2 = sigb;
  }
  else  {
    div2 = 0;
  }
  div3 = sigb << 3;
  divSum = div1 ^ div2 ^ div3;
  divCar = (div1 & div2 | div1 & div3 | div2 & div3) << 1;
  ui57 div = divSum + divCar;

  // Prescale the dividend using the same scaling factor:
  ui56 rem1, rem2, rem3, remSum, remCar;
  if (!sigb[51] && (sigb[50] || !sigb[49])) {
    rem1 = siga << 2;
  }
  else if (!sigb[50] && (sigb[51] || sigb[49])) {
    rem1 = siga << 1;
  }
  else  {
    rem1 = 0;
  }
  if (!sigb[51] && !sigb[50]) {
    rem2 = siga << 2;
  }
  else if ((sigb[51] || sigb[50]) && !sigb[49] || sigb[51] && sigb[50]) {
    rem2 = siga;
  }
  else  {
    rem2 = 0;
  }
  rem3 = siga << 3;
  remSum = rem1 ^ rem2 ^ rem3;
  remCar = (rem1 & rem2 | rem1 & rem3 | rem2 & rem3) << 1;

  // Compare siga and sigb:
  ui53 sigaBar = ~siga;
  ui54 sigCmp = sigb + sigaBar;
```

```
  bool sigaLTsigb = sigCmp[53];

  // Compute 5-bit approximation of scaled dividend:
  ui5 remCarBits, remSumBits;
  bool remCin;
  if (sigaLTsigb) {
    remCarBits = remCar[55:52];
    remSumBits = remSum[55:52];
    remCin = remCar[51] || remSum[51];
  }
  else  {
    remCarBits = remCar[55:53];
    remSumBits = remSum[55:53];
    remCin = remCar[52] || remSum[52];
  }
  ui5 remBits = remCarBits + remSumBits + remCin;

  // q1 = 2 if remBits[4:0] >= 13, otherwise q1 = 1:
  int q1 = remBits[4] || remBits[3] && remBits[2] & (remBits[1] || remBits[0]) ? 2 : 1;

  // Carry vector of R1 and exponent of the quotient:
  ui59 RP = remSum + remCar;
  if (sigaLTsigb) {
    RP <<= 1;
    expDiff--;
  }

  // sum vector of R1:
  ui59 RN = 0;
  if (q1 == 2) {
    RN[57:1] = div;
  }
  else  {
    RN[56:0] = div;
  }
  return <div, RP, RN, expDiff, q1>;
}

// Derive the next quotient digit qi+1 from a 6-bit approximation of the remainder Ri:

iint nextDigit(ui6 remS6) {

  // remS6 >= 13:
  if (!remS6[5] && (remS6[4] || (remS6[3] && remS6[2] && (remS6[1] || remS6[0])))) {
    return 2;
  }

  // remS6 >= 4
   else if (!remS6[5] && (remS6[3] || remS6[2])) {
    return 1;
  }

  // remS6 >= -3
  else if (!remS6[5] ||
          remS6[5] && remS6[4] && remS6[3] && remS6[2] && (remS6[1] || remS6[0])) {
    return 0;
  }

  // remS6 >= -12
  else if (remS6[4] && (remS6[3] || remS6[2])) {
    return -1;
  }
  else  {
    return -2;
  }
}

// Derive the next remainder Ri+1 from the remainder Ri, quotient digit qi+1,
// and divisor:
```

```
<ui59, ui59> nextRem(ui59 RP, ui59 RN, ui59 div, int q, ui2 fmt) {
  ui59 divMult = div;
  switch (q) {
  case 2:
    divMult <<= 1;
    divMult = ~divMult;
    break;
  case 1:
    divMult = ~divMult;
    break;
  case 0:
    divMult = 0;
    break;
  case -1:
    break;
  case -2:
    divMult <<= 1;
  }
  ui59 RP4 = RP << 2;
  ui59 RN4 = RN << 2;
  ui59 sum = RN4 ^ RP4 ^ divMult;
  ui59 car = ~RN4 & RP4 | (~RN4 | RP4) & divMult;
  ui59 car2 = car << 1;
  switch (fmt) {
  case DP:
    RP = car2;
    RP[0] = q > 0;
    RN = sum;
    break;
  case SP:
    RP[58:29] = car2[58:29];
    RP[29] = q > 0;
    RN[58:29] = sum[58:29];
    break;
  case HP:
    RP[58:42] = car2[58:42];
    RP[42] = q > 0;
    RN[58:42] = sum[58:42];
  }
  return <RP, RN>;
}

// Update signed-digit quotient with next digit:

<ui54, ui54> nextQuot(ui54 QP, ui54 QN, int q) {
  QP <<= 2;
  QN <<= 2;
  if (q >= 0) {
    QP[1:0] = q;
  }
  else  {
    QN[1:0] = -q;
  }
  return <QP, QN>;
}

// In each of the three iterations of a cycle, the next quotient
digit and remainder
// (in redundant form) are computed.  The remainder upon
entering the cycle is Ri.
// The quotient digits and remainders computed in the cycle are
qi1, qi2, qi3, Ri1,
// Ri2, Ri3. The remainders are redundantly represented by RPi*
and RNi*.

// In the first iteration, two approximations of Ri1 are
returned along with qi1, RPi1,
// and RNi1:
// (1) a 6-bit sum Ri1S6, which is used in the second iteration
to compute qi2;
```

```
// (2) a 9-bit sum Ri1S9, which is used in the second iteration
in combination with the
//     divisor to compute a 7-bit approximation of Ri2, used
in the third iteration to
//     compute qi3.

<int, ui59, ui59, ui6, ui9> iter1(ui59 RPi, ui59 RNi, ui57 div, ui2 fmt) {
  ui6 RiS6 = RPi[56:51] + ~RNi[56:51] + 1;
  int qi1 = nextDigit(RiS6);
  ui59 RPi1, RNi1;
  <RPi1, RNi1> = nextRem(RPi, RNi, div, qi1, fmt);
  ui6 Ri1S6 = RPi1[56:51] + ~RNi1[56:51] + 1;
  ui9 Ri1S9 = RPi1[56:48] + ~RNi1[56:48] + 1;
  return <qi1, RPi1, RNi1, Ri1S6, Ri1S9>;
}

// In the second iteration, a 7-bit non-redundant approximation of Ri2 is returned
// along with qi2, RPi2, and RNi2:

<int, ui59, ui59, ui7>
iter2(ui59 RPi1, ui59 RNi1, ui6 Ri1S6, ui9 Ri1S9, ui57 div, ui2 fmt) {
  int qi2 = nextDigit(Ri1S6);
  ui59 RPi2, RNi2;
  <RPi2, RNi2> = nextRem(RPi1, RNi1, div, qi2, fmt);
 ui7 divShft7;
  switch (qi2) {
  case 2:
    divShft7 = ~div[55:49];
    break;
  case 1:
    divShft7 = ~div[56:50];
    break;
  case 0:
    divShft7 = 0;
    break;
  case -1:
    divShft7 = div[56:50];
    break;
  case -2:
    divShft7 = div[55:49];
  }
  ui7 Ri2S7 = Ri1S9[6:0] + divShft7 + 1;
  return <qi2, RPi2, RNi2, Ri2S7>;
}

// The third iteration returns qi3, RPi3, and RNi3:

<int, ui59, ui59> iter3(ui59 RPi2, ui59 RNi2, ui7 Ri2S7, ui57 div, ui2 fmt) {
  int qi3 = nextDigit(Ri2S7[6:1]);
  ui59 RPi3, RNi3;
  <RPi3, RNi3> = nextRem(RPi2, RNi2, div, qi3, fmt);
  return <qi3, RPi3, RNi3>;
}

<ui64, ui8> fdiv64(ui64 opa, ui64 opb, ui2 fmt, bool fz, bool dn, ui2 rmode) {

  // Analyze operands and process special cases:
  bool signa, signb;    // operand signs
  ui11 expa, expb;      // operand exponents
  ui52 mana, manb;      // operand mantissas
  Class classa, classb; // operand classes
  ui8 flags = 0;        // exception flags
  tie(signa, expa, mana, classa, flags) = analyze(opa, fmt, fz, flags);
  tie(signb, expb, manb, classb, flags) = analyze(opb, fmt, fz, flags);
  bool sign = signa ^ signb; // sign of quotient

  // Detect early exit:
  if (classa == ZERO || classa == INF || classa == SNAN || classa == QNAN ||
      classb == ZERO || classb == INF || classb == SNAN || classb == QNAN) {
    return specialCase(sign, opa, opb, classa, classb, fmt, dn, flags);
```

```
    }
    else {

      // Detect division by a power of 2:
      bool divPow2 = classa == NORM && classb == NORM && manb == 0;

     // Normalize denormals and compute exponent difference:
      ui53 siga, sigb; // significands
      si13 expDiff;    // exponent difference
      <siga, sigb, expDiff> = normalize(expa, expb, mana, manb, fmt);
      ui57 div;    // non-redundant prescaled divisor
      ui59 RP, RN; // redundant prescaled remainder
      ui54 QP = 0, QN = 0; // redundant quotient
      si13 expQ; // quotient exponent
      int q; // quotient digit

      // Prescale divisor and remainder
      <div, RP, RN, expQ> = prescale(siga, sigb, expDiff);

      ui5 N; // number of cycles in the iterative phase
      if (divPow2) {
        N = 0;
      }
      else {
        switch (fmt) {
        case DP:
          N = 9;
          break;
        case SP:
          N = 4;
          break;
        case HP:
          N = 2;
        }
      }
      for (uint i = 0; i < N; i++) { // ith cycle, consisting of 3 iterations

        // 1st iteration:
        ui6 RS6; // 6-bit approximation of remainder
        ui9 RS9; // 9-bit approximation of remainder
        <q, RP, RN, RS6> = iter1(RP, RN, div, fmt);
        <QP, QN> = nextQuot(QP, QN, q);

        // 2nd iteration:
        ui7 RS7;
        <q, RP, RN, RS7> = iter2(RP, RN, RS6, RS9, div, fmt);
        <QP, QN> = nextQuot(QP, QN, q);

        // 3rd iteration:
        <q, RP, RN> = iter3(RP, RN, RS7, div, fmt);
        <QP, QN> = nextQuot(QP, QN, q);
      }

     // Assimilate quotient:
     ui53 Qtrunc, Qinc; // Non-redundant quotient and incremented quotient
      bool stk;     // sticky bit
      if (divPow2) {
        Qtrunc = mana << 1;
        stk = 0;
      }
      else {
        <Qtrunc, Qinc, stk> = computeQ(QP, QN, RP, RN, fmt, false);
      }

      // Round:
      ui53 Qrnd, QrndDen;
      bool inx, inxDen;
      <Qrnd, inx, QrndDen, inxDen> = rounder(Qtrunc, Qinc, stk, sign, expQ, rmode, fmt);

      // Compute exceptions and assemble final result:
```

```
    return final(Qrnd, inx, QrndDen, inxDen, sign, expQ, rmode, fz, fmt, flags);
  }
}
```

E Multi-Precision Radix-4 Square Root

```
// Zero, infinity, NaN, or negative operand:

<ui64, ui8> specialCase(bool signa, ui64 opa, Class classa, ui2 fmt, bool dn, ui8 flags) {
  ui64 D = 0;
  ui64 aTrunc, manMSB, defNaN, zero = 0;
  switch (fmt) {
  case DP:
    aTrunc = opa[63:0];
    zero[63] = signa;
    defNaN = 0x7FF8000000000000;
    manMSB = 0x8000000000000;
    break;
  case SP:
    aTrunc = opa[31:0];
    zero[31] = signa;
    defNaN = 0x7FC00000;
    manMSB = 0x400000;
    break;
  case HP:
    aTrunc = opa[15:0];
    zero[15] = signa;
    defNaN = 0x7E00;
    manMSB = 0x200;
    break;
  }
  if (classa == SNAN) {
    D = dn ? defNaN : aTrunc | manMSB;
    flags[IOC] = 1;
  }
  else if (classa == QNAN) {
    D = dn ? defNaN : aTrunc;
  }
  else if (classa == ZERO) {
    D = zero;
  }
  else if (signa) {
    D = defNaN;
    flags[IOC] = 1;
  }
  else  {
    D = aTrunc;
  }
  return <D, flags>;
}

// Normalize denormal operand and compute predicted result exponent:

<ui53, si13, ui11> normalize(si13 expa, ui52 mana, ui2 fmt) {
  ui53 siga = 0;
  uint bias;
  switch (fmt) {
  case DP:
    siga = mana;
    bias = 0x3FF;
    break;
  case SP:
```

```
    siga[51:29] = mana;
    bias = 0x7F;
    break;
  case HP:
    siga[51:42] = mana;
    bias = 0xF;
  }
  if (expa == 0) {
    ui6 clz = CLZ53(siga);
    siga <<= clz;
    expa = 1 - clz;
  }
  else  {
    siga[52] = 1;
  }
  ui12 expQ = expa + bias;
  return <siga, expa, expQ[11:1]>;
}

// Power of 2:

<ui64, ui8> sqrtPow2(ui11 expQ, bool expOdd, ui2 rmode, ui2 fmt) {
  ui64 D = 0;
  ui8 flags = 0;
  uint manWidth;
  ui52 manSqrt2;
  switch (fmt) {
  case DP:
    manWidth = 52;
    manSqrt2 = rmode == rmodeNear || rmode == rmodeUP ? 0x6A09E667F3BCD : 0x6A09E667F3BCC;
    break;
  case SP:
    manWidth = 23;
    manSqrt2 = rmode == rmodeUP ? 0x3504F4 : 0x3504F3;
    break;
  case HP:
    manWidth = 10;
    manSqrt2 = rmode == rmodeUP ? 0x5A9 : 0x5A8;
    break;
  }
  if (!expOdd) {
    D = manSqrt2;
    flags[IXC] = 1;
  }
  D[manWidth + 10:manWidth] = expQ;
  return <D, flags>;
}

// First iteration:

<ui59, ui59, ui54, int, uint> firstIter(ui53 siga, bool expOdd) {
  ui59 RP = 0, RN = 0;
  ui54 QN = 0;
  int q;
  uint i;
  if (expOdd) {
    // x = siga/4 = .01xxx...
    // R0 = x - 1 = 1111.01xxx...
    // RP = 4*R0 = 1101.xxx...
    RP[58:56] = 6;
    RP[55:3] = siga;
    if (siga[51]) {
      // -5/2 <= 4*R0 < -2
      q = -1;
      QN[53:52] = 1; // .01000...
      // R1 = 4*R0 - (-1) * (2*Q0 + (-1)/4) = 4*R0 + 7/4
      // RN = -7/4 = 1110.0100..
      RN[58:53] = 0x39;
```

```
      i = 4;
    }
    else  {
      // 4*R0 < -5/2
      q = -2;
      QN[53:52] = 2; // .10000...
      // R1 = 4*R0 - (-2) * (2*Q0 + (-2)/4) = 4*R0 + 3
      // RN = -3 = 1101.00...
      RN[58:55] = 0xD; // 1110.0100...
      i = 0; // Q1 = 0.1000
    }
  }
  else  { // expa even
    // x = siga/2 = .1xxx...
    // R0 = x - 1 = 1111.1xxx...
    // RP = 4*R0 = 111x.xx...
    RP[58:57] = 3;
    RP[56:4] = siga;
    if (siga[51]) {
      // -1 <= 4*R0 < 0
      q = 0;
      // QN = 0
      // R1 = 4*R0 = RP, RN = 0
      i = 8; // Q1 = 1.0000
    }
    else  {
      // -2 <= 4*R0 < -1
      q = -1;
      QN[53:52] = 1;// .01000...
      // R1 = 4*R0 - (-1) * (2*Q0 + (-1)/4) = 4*R0 + 7/4
      // RN = -7/4 = 1110.0100...
      RN[58:53] = 0x39;
      i = 4; // Q1 = 0.1100
    }
  }
  return <RP, RN, QN, q, i>;
}

//   Derive the next quotient digit q_(j+1) from the root interval i and remainder R_j:

int nextDigit(ui59 RP, ui59 RN, uint i, uint j) {
  ui59 RP4 = RP << 2, RN4 = RN << 2;
  ui8 RS8 = RP4[58:51] + ~RN4[58:51] + (RP4[50] || !RN4[50]);
  si7 RS7 = RS8[7:1];
  si7 mp2, mp1, mz0, mn1;
  switch (i) {
  case 0: mp2 = 12; mp1 = 4; mz0 = -4; mn1 = j == 1 ? -11 : -12; break;
  case 1: mp2 = j == 2 ? 15 : 13; mp1 = 4; mz0 = -4; mn1 = -13; break;
  case 2: mp2 = 15; mp1 = 4; mz0 = -4; mn1 = -15; break;
  case 3: mp2 = 16; mp1 = 6; mz0 = -6; mn1 = -16; break;
  case 4: mp2 = 18; mp1 = 6; mz0 = -6; mn1 = -18; break;
  case 5: mp2 = 20; mp1 = 8; mz0 = -6; mn1 = -20; break;
  case 6: mp2 = 20; mp1 = 8; mz0 = -8; mn1 = -20; break;
  case 7: mp2 = 22; mp1 = 8; mz0 = -8; mn1 = -22; break;
  case 8: mp2 = 24; mp1 = 8; mz0 = -8; mn1 = -24;
  }
  int q;
  if (RS7 >= mp2) {
    q = 2;
  }
  else if (RS7 >= mp1) {
    q = 1;
  }
  else if (RS7 >= mz0) {
    q = 0;
  }
  else if (RS7 >= mn1) {
    q = -1;
```

```
  }
  else  {
    q = -2;
  }
  return q;
}

// Derive the next remainder R_(j+1) from the remainder R_j and the quotient digit q_(j+1):

<ui59, ui59> nextRem(ui59 RP, ui59 RN, ui54 QP, ui54 QN, int q, uint j, ui2 fmt) {

  // Dcar - Dsum = D = 2 * Q_j + 4^(-(j+1)) * q_(j+1):
  ui59 Dcar = 0, Dsum = 0;
  Dcar[56] = 1; // integer bit, implicit in QP
  Dcar[55:2] = QP;
  Dsum[55:2] = QN;
  if (q > 0) {
    Dcar[53 - 2 * j + 1:53 - 2 * j] = q;
  }
  else if (q < 0) {
    Dsum[53 - 2 * j + 1:53 - 2 * j] = -q;
  }

  // DQcar - DQsum = -q_(j+1) * D:
  ui59 DQcar, DQsum;
  switch (q) {
  case 1:
    DQcar = Dsum;
    DQsum = Dcar;
    break;
  case 2:
    DQcar = Dsum << 1;
    DQsum = Dcar << 1;
    break;
  case -1:
    DQcar = Dcar;
    DQsum = Dsum;
    break;
  case -2:
    DQcar = Dcar << 1;
    DQsum = Dsum << 1;
  }

  // RP4 - RN4 = 4 * R_j:
  ui59 RP4 = RP << 2, RN4 = RN << 2;

  // car1 - sum1 = RP4 - RN4 + DQcar = 4 * R + DQcar:
  ui59 sum1 = RN4 ^ RP4 ^ DQcar;
  ui59 car1 = (~RN4 & RP4 | (~RN4 | RP4) & DQcar) << 1;
  if (fmt == HP) {
    car1[42] = 0;
  }
  else if (fmt == SP) {
    car1[29] = 0;
  }

  // car2 - sum2 = car1 - sum1 - DQsum
  //             = 4 * R_j + DQcar - DQsum
  //             = 4 * R_j - q_(j+1) * D
  //             = 4 * R_j - q_(j+1) * (2*Q<_j + 4^(-(j+1)) * q_(j+1)):
  ui59 sum2 = sum1 ^ car1 ^ ~DQsum;
  ui59 car2 = (~sum1 & car1 | (~sum1 | car1) & ~DQsum) << 1;
  if (q == 0) {
    return <RP4, RN4>;
  }
  else  {
    switch (fmt) {
    case DP:
```

```
      car2[0] = 1;
      RP = car2;
      RN = sum2;
      break;
    case SP:
      car2[29] = 1;
      RP[58:29] = car2[58:29];
      RN[58:29] = sum2[58:29];
      break;
    case HP:
      car2[42] = 1;
      RP[58:42] = car2[58:42];
      RN[58:42] = sum2[58:42];
    }
    return <RP, RN>;
  }
}

// Update signed-digit quotient with next digit q_(j+1):

<ui54, ui54> nextRoot(ui54 QP, ui54 QN, int q, uint j) {
  if (q > 0) {
    QP[52 - 2 * j + 1:52 - 2 * j] = q;
  }
  else if (q < 0) {
    QN[52 - 2 * j + 1:52 - 2 * j] = -q;
  }
  return <QP, QN>;
}

<ui64, ui8> fsqrt64(ui64 opa, ui2 fmt, bool fz, bool dn, ui2 rmode) {
  bool signa;     // operand signs
  ui11 expa;      // operand exponents
  ui52 mana;      // operand mantissas
  Class classa;   // operand classes
  ui8 flags = 0;  // exception flags
  <signa, expa, mana, classa> = analyze(opa, fmt, fz, flags);

  // Detect early exit:
  if (classa == ZERO || classa == INF || classa == SNAN || classa == QNAN || signa) {
    return specialCase(signa, opa, classa, fmt, dn, flags);
  }
  else  {
    bool expInc = classa == NORM && rmode == rmodeUP;

    // Normalize denormal and compute predicted result exponent:
    ui53 siga;    // significand
    si13 expShft; // adjusted exponent
    ui11 expQ;    // predicted result exponent
    <siga, expShft, expQ> = normalize(expa, mana, fmt);

    bool expOdd = expShft[0]; // parity of adjusted exponent

    if (classa == NORM && mana == 0) { // power of 2
      return sqrtPow2(expQ, expOdd, rmode, fmt);
    }
    else  {
       ui59 RP, RN;  // redundant remainder
      ui54 QP, QN;  // redundant root
      int q;        // root digit;
      uint i;       // root interval, 0 <= i <= 8

      // First iteration:
      <RP, RN, QN, q> = firstIter(siga, expOdd);
      QP = 0;
      expInc &= QN == 0;

      ui5 N; // number of iterations
```

```
        switch (fmt) {
        case DP:
          N = 27;
          break;
        case SP:
          N = 13;
          break;
        case HP:
          N = 6;
        }
        for (uint j = 1; j < N; j++) {
          q = nextDigit(RP, RN, i, j);
          if (j == 1) {
            i = i + q;
          }
          <RP, RN> = nextRem(RP, RN, QP, QN, q, j, fmt);
          <QP, QN> = nextRoot(QP, QN, q, j);
          expInc &= j < N - 1 ? q == 0 : fmt == SP ? q == -2 : q == -1;
        }
        ui11 expRnd = expInc ? expQ + 1 : expQ;

        // Assimilate root:
        switch (fmt) { // first move to low bits
        case HP:
          QP = QP[53:42];
          QN = QN[53:42];
          break;
        case SP:
          QP = QP[53:28];
          QN = QN[53:28];
          break;
        }
        ui53 Qtrunc, Qinc; // Non-redundant quotient and incremented quotient
        bool stk;     // sticky bit
        <Qtrunc, Qinc, stk> = computeQ(QP, QN, RP, RN, fmt, true);

        // Round:
        ui53 Qrnd, QrndDen;
        bool inx, inxDen;
        <Qrnd, inx, QrndDen, inxDen> = rounder(Qtrunc, Qinc, stk, 0, expRnd, rmode, fmt);


        // Compute exceptions and assemble final result:
        return final(Qrnd, inx, QrndDen, inxDen, 0, expRnd, rmode, fz, fmt, flags);
      }
    }
  }
```

Bibliography

1. Boldo, S., Melquiond, G.: Emulation of a FMA and correctly rounded sums: Proved algorithms using rounding to odd. Technical Report HAL Id: 00080427, Inria (2010)
2. Booth, A. D.: A signed binary multiplication technique. Quarterly Journal of Mechanics and Applied Mathematics **4** (1951)
3. Burks, A. W., Goldstine, H. H., von Neumann, J.: Preliminary discussion of the logical design of an electronic computing instrument. In: B. Randell (ed.) The Origins of Digital Computers. Springer, Berlin (1982)
4. Cocke, J., Sweeney, D. W.: High speed arithmetic in a parallel device (1957). Available at http://archive.computerhistory.org/resources/text/IBM/Stretch/pdfs/06-08/102632302.pdf
5. Ercegovac, M. D., Lang, T.: Division and Square Root Digit-Recurrence Algorithms and Implementations. Kluwer Academic Publishers (1994)
6. Gamboa, R.: Real-valued algorithms in ACL2. Ph.D. thesis, University of Texas at Austin, Department of Computer Sciences (1999)
7. Harrison, J.: Formal verification of IA-64 division algorithms. In: M. Aagaard, J. Harrison (eds.) Theorem Proving in Higher Order Logics: 13th International Conference, *Lecture Notes in Computer Science*, vol. 1869. Springer-Verlag (2000)
8. Institute of Electrical and Electronic Engineers: IEEE standard for floating point arithmetic, Std. 754-1985 (1985)
9. Institute of Electrical and Electronic Engineers: IEEE standard for floating point arithmetic, Std. 754-2008 (2008)
10. Josuttis, N. M.: The C++ Standard Library: A Tutorial and Reference, 2nd edn. Pearson Education, Inc. (2012)
11. Kaufmann, M., Manolios, P., Moore, J S. (eds.): Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Press (2000)
12. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Press (2000)
13. Kaufmann, M., Moore, J S.: ACL2 web site. http://www.cs.utexas.edu/users/moore/acl2/
14. Kline, M.: Mathematical Thought from Ancient to Modern Times. Oxford University Press (1972)
15. Koren, I.: Computer Arithmetic Algorithms, 2nd edn. A. K. Peters (1993)
16. Levine, J.: Flex and Bison. O'Reilly Media (2009)
17. Lutz, D. R.: Fused multiply-add architecture comprising separate early-normalizing multiply and add pipelines. In: 20th IEEE Symposium on Computer Arithmetic (2011). Available at https://www.computer.org/csdl/proceedings/arith/2011/9457/00/05992117.pdf

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1

18. Markstein, P. W.: Computation of elementary functions on the IBM RISC System/6000 processor. IBM Journal of Research and Development **34**(1) (1990)
19. Mentor Graphics Corp.: Algorithmic C datatypes. Available at https://www.mentor.com/hls-lp/downloads/ac-datatypes
20. Mentor Graphics Corp.: Sequential logic equivalence checker. https://www.mentor.com/products/fv/questa-slec
21. Mizouni, R., S, Tahar, Curzon, P.: Hybrid verification integrating HOL theorem proving with MDG model checking. Microelectronics Journal **37**(11) (2006)
22. Moore, J S., Lynch, T., Kaufmann, M.: A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating point division algorithm. IEEE Trans. Comput. **47**(9) (1998)
23. O'Leary, J. W., Russinoff, D. M.: Modeling algorithms in SystemC and ACL2. In: ACL2 2014: 12th International Workshop on the ACL2 Theorem Prover and its Applications. Vienna (2014). Available at http://www.russinoff.com/papers/masc.html
24. Pineiro, J.-A., Oberman, S. F., Muller, J.-M., Bruguera, J. D.: High-speed function approximation using a minimax quadratic interpolator. IEEE Trans. Comput. **54**(3) (2005). Available at http://perso.ens-lyon.fr/jean-michel.muller/QuadraticIEEETC0305.pdf
25. Ray, S., Sumners, R.: Combining theorem proving with model checking through predicate abstraction. IEEE Design and Test Special Issue on Advances in Functional Validation through Hybrid Techniques (2007)
26. Robertson, J. E.: A new class of digital division methods. IRE Transactions on Electronic Computers **EC-7** (1958)
27. Russinoff, D. M.: A mechanically checked proof of IEEE compliance of the AMD-K7 floating point multiplication, division, and square root instructions. London Mathematical Society Journal of Computation and Mathematics **1**, 148–200 (1998). Available at http://www.russinoff.com/papers/k7-div-sqrt.html
28. Russinoff, D. M.: A mechanically checked proof of IEEE compliance of the AMD-K5 floating point square root microcode. Formal Methods in System Design **14**, 75–125 (1999). Available at http://www.russinoff.com/papers/fsqrt.html
29. Russinoff, D. M.: A case study in formal verification of register-transfer logic with ACL2: the floating point adder of the AMD Athlon processor. In: Formal Methods in Computer-Aided Design (2000). Available at http://www.russinoff.com/papers/fadd.html
30. Russinoff, D. M.: A mechanically verified commercial SRT divider. In: D.S. Hardin (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 23–63. Springer (2010)
31. Russinoff, D. M.: Computation and formal verification of SRT quotient and square root digit selection tables. IEEE Trans. Comput. **62**(5), 900–913 (2013). Available at http://www.russinoff.com/papers/srt8.html
32. Russinoff, D. M., Flatau, A.: RTL verification: a floating-point multiplier, chap. 13, pp. 201–231. In: Kaufmann et al. [11] (2000). Available at http://www.russinoff.com/papers/acl2.pdf
33. Seger, C. H., Jones, R. B., O'Leary, J. W., Melham, T., Aagaard, M. D., Barrett, C., Syme, D.: An industrially effective environment for formal hardware verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **24**(9), 1381–1405 (2005)
34. Synopsys, Inc.: Hector. http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/hector.aspx
35. Tocher, K. D.: Techniques of multiplication and division for automatic binary computers. Quarterly Journal of Mechanics and Applied Mathematics **2** (1958)

Index

A
Addition, 147–163, 279–307
Algorithmic C, 237, 239, 243–245
Arm instructions, 233
  floating-point status and control register, 234
  post-computation exceptions, 236
  pre-computation exceptions, 235–236

B
Binary representation, 17
Bit extraction, 17, 24–28
Bit slice, 17–24
Bit vector, 17–37
  width, 17
Booth encoding, 165
  of carry-save sum, 173–176
  radix-2, 166–167
  radix-4, 167–173
  radix-8, 179–182
  statically encoded arrays, 176–179

C
Carry-look-ahead adder, 154
Carry-save adder, 150
Carry-save representation, 173
Ceiling, 5–6
Compressor, 3:2, 150
Concatenation, 28–32
Congruence, 8
Constant injection, 92, 122
Correct rounding principle, 51, 219

D
Denormal encoding, 69–73
Digit recurrence, 183
Division, 183–191, 203–217, 309–329

E
Exponent, 53
Exponent bias, 65

F
Far path, 156
Fixed-point format, 37, 35–37
Floating-point format
  double extended precision, 64
  double precision, 64
  explicit, 63
  half precision, 64
  implicit, 63
  single precision, 64
Floating-point number
  decomposition, 53–55
  exactness, 56–61
Floor, 3–5
FMA, 204, 267–268, 306
Full adder, 149

H
Half adder, 147

I
IEEE compliance, 51, 63, 117, 135, 219
Infinity, 74

© Springer Nature Switzerland AG 2019

D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design*,
https://doi.org/10.1007/978-3-319-95513-1

L
Largest positive normal, 68
Leading zero anticipation, 155–160
Logical operations, 39–50
  and, 39
  complement, 46
  exclusive or, 39
  inclusive or, 39

M
Minimax quadratic interpolation, 205
Modulus, 6–12
Multiplication, 165–182, 253–277

N
NaN, 74
Near path, 156
netlist, 1
Newton-Raphson approximation, 204
Normal encoding, 65–69
Numerical encoding, 69

O
Overflow, 78

P
Propagate and generate vectors, 147, 157
Pseudo-denormal encoding, 69

Q
QNaN, 74

R
Reciprocal approximation, 205
Redundant representation, 34, 187
Ripple-carry adder, 152
Rounding, 118–126
  away from zero, 86–93
  decomposable, 78
  denormal, 127–134
  double rounding, 78, 110
  nearest, 93–109
  odd, 109–117
  rounding mode, 77
  symmetric, 78
  toward zero, 79–86
  unbiased, 78, 100
  von Neumann, 109

S
Sign, 53
Signed integer, 32–35
Sign extension, 33
Significand, 53
SIMD instructions, 219
Smallest positive denormal, 72
Smallest positive normal, 68
SNaN, 74
Square root, 135–144, 191–202, 331–343
SRT division and square root, 183–202
SSE instructions
  control and status register, 221–222
  post-computation exceptions, 225–226
  pre-computation exceptions, 223–224

T
Trailing zero anticipation, 163, 160–163
Truncation, 12–15
Two's complement, 33

U
Underflow, 78
Unsupported encoding, 65

X
x87 instructions
  control word, 227–228
  post-computation exceptions, 231
  pre-computation exceptions, 230
  status word, 228–229

Z
Zero encoding, 69