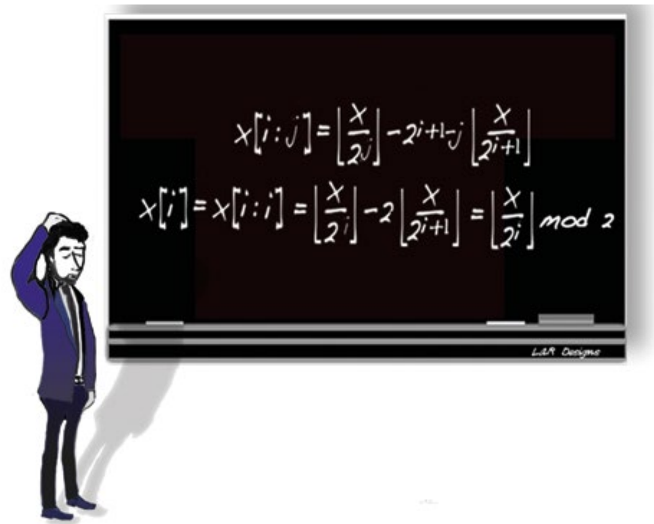
David M. Russinoff




# 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