A Mechanically Checked Proof of the AMD5$_K$86™ Floating-Point Division Program

J Strother Moore, Thomas W. Lynch, *Member*, *IEEE*, and Matt Kaufmann

Abstract—In this article, we report the successful application of a mechanical theorem prover to the problem of verifying the division microcode program used on the AMD5$_K$86 microprocessor. The division algorithm is an iterative shift and subtract type. It was implemented using floating-point microcode instructions. As a consequence, the floating quotient digits have data dependent precision. This breaks the constraints of conventional SRT division theory. Hence, an important question was whether the algorithm still provided perfectly rounded results at 24, 53, or 64 bits. The mechanically checked proof of this assertion is the central topic of this paper. The proof was constructed in three steps. First, the divide microcode was translated into a formal intermediate language. Then, a manually created proof was transliterated into a series of formal assertions in the ACL2 dialect. After many expansions and modifications to the original proof, the theorem prover certified the assertion that the quotient will always be correctly rounded to the target precision.

Index Terms—Division, verification, computer arithmetic, formal verification, theorem proving.

1 Introduction

The AMD5$_K$86 floating-point unit major blocks include two full width barrel shifters, a full width significand adder, a 32 by 32 bit multiplier, and a rounder. The floating-point unit data path accommodates the common IEEE std. 754 formats [39]. Microcode operations may use extended range versions of these formats. All complex instructions are microcoded. The microcode instructions are vertically integrated and resemble assembly level floating-point instructions.

The divide microcode consists of the familiar shift and subtract loop. This is repeated four times, with each iteration producing an extended range single precision floating quotient digit. The first remainder is set to the value of the double extended dividend. Correct rounding occurs naturally as a consequence of performing the last iteration. The quotient guessing technique is the "arithmetic model" reported by Atkins [7], where a single precision quotient guess is formed by approximating the reciprocal of the divisor and, then, multiplying by the partial remainder. As was done by Atkins, we use Newton-Raphson for forming the reciprocal in accordance with Wallace [8] and Farrari [9]. The seed for the NR iterations is obtained using the method described by Schulte in [10]. The algorithm is described in detail in Section 3.

More information on seed tables can be found in [2]. Briggs and Matula reported an SRT divider using the arithmetic model in [11]. Briggs and Matula's divider uses a rectangular multiplier to efficiently handle the extended precision caused by the quotient digit times divisor multiply. Rectangular multiplier techniques are extended in [13]. The algorithm to be presented is related to the high radix method described by Wong and Flynn in [14]. A general discussion of division can be found in [29] and [28]. A brief survey of design decisions in division algorithms is presented in [29].

The AMD5$_K$86 division program differs from the SRT techniques since each floating-point quotient digit, FQD, has a data dependent "fractional part" which will not appear in a corresponding SRT algorithm of the same radix. Nor does the AMD5$_K$86 division program use a rectangular multiplier. Instead, where extra precision must be maintained, variations of the techniques presented by Priest [15] are used.

As the AMD5$_K$86 divide progresses, the FQDs may float apart or overlap, due to floating-point normalization and reciprocal estimation errors. As the FQDs float, so do subsequent partial remainders. This behavior is similar to that caused by the zero skipping algorithm of Wilson [18]. The basic principle which guided the design of the AMD5$_K$86 division program was that the partial remainders should exactly correspond to the quotient guesses. A simple floating-point forward error analysis [16], [17], [19] was employed to find the error in the remainder while assuming the quotient guess was exact. This error term was then forced to zero by design, as further described in later sections.

The final step in our division algorithm is the floating-point summation of the first FQD to the extended precision accumulation of the other FQDs. The accumulation of FQDs uses a special rounding mode, called "sticky," which is the "unbiased" rounding mode described by Sterbenz [17]. In this rounding mode, the lsb of the result plays the role of the sticky bit, and no further rounding is done.

Denormal numbers will be created in the final step if the target format does not have sufficient exponent range to

- *J S. Moore is with the Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188. E-mail: moore@cs.utexas.edu.*
- *T.W. Lynch is with THS, 701 Brazos, Suite 500, Austin, TX 78759. E-mail: lynch@ths.com.*
- *M. Kaufmann is with EDS, 98 San Jacinto Blvd., Suite 500, Austin, TX 78701.*

Manuscript received March 1996.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106943.

hold a very small normalized result. This occurs naturally as part of the functionality of the adder. Such cases are covered in the proof by virtue of the fact that the final theorem holds for an n bit precise quotient, where n may be less than or equal to the target precision. As set forth in the lemmas on rounding given later, double rounding is sound when the first rounding mode is *sticky*, and the second rounding reduces the precision still further, and by a significant amount.

This work was part of the project to verify the AMD5$_K$86 numerical code which was described in [5]. Following the methodology given in that paper, the proofs here are broken into two parts. The first part verifies the correctness of the algorithm with arbitrary precision partial remainders, while the second part assures correct behavior with finite precision and range. A third aspect is not discussed here, and that is the verification of the algorithm when the operands are NaNs, Infinities, or signed Zeros. This was verified through exhaustive testing, not proof.

Moore and Kauffman applied the *ACL2* theorem prover to the task of checking the numerical proof. This tool is the successor to the Boyer-Moore theorem prover, Nqthm, and its interactive enhancement [21], [20]. The prover attempts to find proof of conjectures submitted by the user based on a collection of already proven lemmas. Hence, the system's behavior is determined by the lemmas it has already proved. The informed user can lead the system to difficult proofs by the selection of appropriate small lemmas. In the context of a project as ambitious as our division proof, it is best to think of ACL2 as a *proof checker*. ACL2 is described in more detail in [32], [33].

Other tools with similar intent exist. For example, internal tools at INMOS were used for the formal verification of microcode on the IMS T800, as discussed by Shephard [1]. PVS [27] was used by Srivas and Miller [2] for verification on the AAMP5 microprocessor. Other work has been done using HOL [3]. Hunt and Brock report the use of Nqthm in the verification of the FM9001 microprocessor [31]. And Coupet-Grimal and Jakubiec discuss the application of *Coq* to verifying arithmetic circuits such as a comparator [6].

Part of ANSI/IEEE-854 [40] is formalized in [36] by Miner using PVS. A few straightforward lemmas about rounding are shown, such as the fact that truncation produces a number of no greater absolute value. However, no mechanically checked proofs of floating-point algorithms are presented.

More recently, Miner used his PVS formalization of ANSI/IEEE-854 in a mechanically checked proof that a parameterized subtractive division algorithm is IEEE compliant [37]. SRT division is an example of a subtractive division algorithm in the class handled by Miner's work. However, the AMD5$_K$86 algorithm is not in this class of algorithms, as it uses nonintegral quotient digits. In a generalization of this work [41], Miner proves that the rational version of the algorithm converges toward the actual quotient. Then, he formalizes an implementation of IEEE rounding, proves it correct with respect to the standard, and shows how the division algorithm can be used to generate sufficient input to the rounder. Along the way, he mechanically proves many lemmas, like those in Subsection 6.2.

There have been several mechanically checked proofs of the SRT division algorithm reported in the literature. In [25], Bryant reports on the use of OBDD techniques to verify certain invariants on a radix-4 SRT division algorithm. Similar work has been done by Clarke, as well as by Clarke et al. (private communication). In [41], Rueß et al. report on the use of the PVS system to verify that a radix r SRT division algorithm divides.

Remarkable proof work continues at AMD [43] under Rusinoff.

2 Formalization and Notation

Our mechanical proof of the correctness of this microcode was carried out with ACL2, "A Computational Logic for Applicative Common Lisp." The ACL2 environment consists of an input language, an interpreter, a base set of knowledge about the ACL2 language (called ACL2 axioms), a theorem prover, and a collection of proven lemmas. The ACL2 axioms describe relationships among the fundamental data types: **Symbols**, **Rationals and Integers**, **Lists**, and **Strings**. The ACL2 theorem prover is capable of using simple inductive and deductive reasoning to attempt certification of the truth of ACL2 language statements. Such statements must be based on the ACL2 axioms and the collection of proven lemmas. Certified statements may be included in the collection of proven lemmas. The ACL2 language is a subset of Common Lisp [44], [45], hence, an expression such as $x \times 2^{i+1}$ is formulated as `(* x (expt 2 (+ i 1)))`. This gives statements in the ACL2 language a dual usage. For example, because the ACL2-based coding of the division program (given in Section 3) is a LISP program, it can be executed; and, because it is also a formal logical description, it can be used as input to the ACL2 theorem prover.

2.1 Floating-Point

Floating-point numbers are built from the fundamental types as follows:[1] Every nonzero rational number x can be uniquely represented in the form $\sigma \times s \times 2^e$ where

- $\sigma \in \{+1, -1\}$,
- s is a rational and $1 \le s < 2$, and
- e is an integer.

We call σ the *sign*, s the *significand*, and e the *exponent* of x. We define the unary function symbols σ, s, and e for accessing the corresponding components of x. We sometimes write σ_x, s_x, and e_x, instead of the more formal $\sigma(x)$, $s(x)$, and $e(x)$. We make the conventions that the sign of 0 is +1, the significand is 0, and the exponent is 0. When we say "s is a significand," we mean $s = 0$ or $1 \le s < 2$.

Note that our notions of significand and exponent are defined for all rationals, not just for those, say, with finite binary expansions. Of course, if a rational has an infinitely repeating binary expansion, then its significand does also. For example, $1/3 = 0.010101..._2$ has a significand of $4/3 = 1.010101..._2$ and an exponent of -2, since $1/3 = 1 \times 4/3 \times 2^{-2}$ and $1 \le 4/3 < 2$. It is helpful to think of significands as being in "normal" form and exponents as being appropriate for normalized significands.

1. We use traditional notation in this paper instead of the ACL2 language statements, as the latter is probably unfamiliar to the reader.

To truncate a significand to i bits, we use:

$$\text{truncn}(s, i) = \frac{\lfloor s \times 2^{i-1} \rfloor}{2^{i-1}}.$$

We say x is an *m,,n normalized floating-point number* if and only if x is rational, truncn(s_x, n) = s_x, and $2 - 2^{m-1} \le e_x \le 2^{m-1} - 1$. We say x is an *m,,n floating-point number* if and only if x is a rational number, truncn(s_x, n) = s_x, and $2 - n - 2^{m-1} \le e_x \le 2^{m-1} - 1$.

We define MinN(m) and MaxN(m, n) to be the minimum and maximum normalized numbers in the m,,n format. We define MinD(m, n) to be the minimum denormal number in the associated format, i.e., $2^{3-n-2^{m-1}}$.

Our notion of the *m,,n normalized* floating-point numbers does not include the NaNs, infinities and signed zeroes of the standard. Our notion of the m,,n floating-point numbers includes all of the standard's normalized and denormal numbers in the corresponding format, as well as additional rationals. The excess comes from the fact that the m,,n formats do not truncate precision off of small numbers. Hence, the set of denormals are a subset of the set of tiny m,,n numbers. By using this broader notion, we simplify and strengthen our theorem. This does not affect our proof because denormals may only occur on input and output of the divide program due to the use of extended range intermediate values. Thus, the AMD5$_K$86 applies the algorithm to a subset of the numbers for which we prove it correct, namely the standard's normalized and denormal numbers.

A final basic concept is that of the ith bit of a significand. We index the bits by the power of 2 indicated by their positions. Because we are thinking here of s as a significand, we know $1 \le s < 2$ and, hence, all bit numbers are nonpositive. Thus, bit 0 is the bit in the ones place, bit –1 is the bit in the 1/2 place, etc. It is useful to remember that the least significant bit in an n bit significand is at position $1 - n$.

$$\text{bitn}(s, i) = \begin{cases} 1 & \text{if } \lfloor s \times 2^{-i} \rfloor \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

2.2 Rounding Styles

A fundamental notion is that of rounding a rational to a given precision. Various rounding "styles" are employed, e.g., round toward 0, away from 0, etc. We define these ideal mathematical notions here. We later restrict and elaborate them (e.g., with denormals or exceptions) to deal with the finiteness of any given representation format.

Six rounding styles are mentioned in this paper. We start with the three that are mentioned explicitly in the division algorithm, **`trunc`**, **`away`**, and **`sticky`**.

$$\text{trunc}(x, i) = \sigma_x \times \text{truncn}(s_x, i) \times 2^{e_x}$$

$$\text{away}(x, i) = \sigma_x \times \text{awayn}(s_x, i) \times 2^{e_x}$$

$$\text{sticky}(x, i) = \sigma_x \times \text{stickyn}(s_x, i) \times 2^{e_x}.$$

Each is defined in terms of an analogous operation on the significand which is then signed and scaled for the general case. We have already defined truncn. awayn(s, i) is:

$$\begin{cases} s & \text{if truncn}(s, i) = s \\ \text{truncn}(s, i) + 2^{1-i} & \text{otherwise} \end{cases}$$

while stickyn(s, i) is:

$$\begin{cases} \text{truncn}(s, i) & \text{if truncn}(s, i) = s \text{ or bitn}(s, 1 - i) = 1 \\ \text{awayn}(s, i) & \text{otherwise.} \end{cases}$$

In addition to the three styles above, we prove that the division algorithm supports three others: round to nearest, to positive infinity, and to negative infinity, denoted by nearest(x, i), posinf(x, i), and neginf(x, i). We omit their definitions here for brevity.

It is convenient to define the function roundx so that, when s is the name of one of the six styles mentioned, roundx(s, x, i) is just s(x, i), e.g., roundx(**`sticky`**, x, i) is sticky(x, i).

2.3 Rounding Modes

The ideal rounding styles above make no provisions for the finiteness of the representation: The rational x is rounded to another rational of the given precision, i. We now deal with finiteness.

A *rounding mode* is a 4-tuple, [s m n uvp], where $s \in$ {**`trunc`**, **`away`**, **`sticky`**, **`nearest`**, **`posinf`**, **`neginf`**}, m and n are integers, $1 < m$, $0 < n \le 64$, and $uvp \in$ {**`unmask-uv`**, **`mask-uv`**}.

The AMD5$_K$86's hardware rounder is modeled as a function, round. It takes a rational, x, and a user-specified rounding mode, [s m n uvp]. The last component of the mode determines the rounder's behavior on underflow. round returns the rounded x, one of the symbols **`overflow-behavior`**, or **`underflow-unmasked-behavior`**, or a pair containing an inexact denormal and the symbol **`uv-flag`**. The last case stems from supporting the gradual underflow scheme, where underflow is signaled when the tiny result is imprecise.

The definition of round(x, [s m n uvp]) is as follows: Let y be sticky(x, 66). This is our formalization of the fact that the data path to the rounder provides a normalized number in 17,,64 format together with two additional bits ("round" and "sticky"). Let z be the user-specified round of y, roundx(s, y, n). If uvp is **`unmask-uv`**, round returns **`overflow-behavior`** if MaxN(m, n) < $|z|$, **`unmasked-underflow-behavior`** if $|z|$ < MinN(m), and z, otherwise. In the case that uvp is **`mask-uv`**, the situation is more complicated. Let n' be $n - (2 - 2^{m-1})$ + e(y). Let y' be the user-specified style round of y to the possibly lower precision n', roundx(s, y, n'). If MaxN(m, n) < $|z|$, round returns **`overflow-behavior`**. If $|z|$ < MinN(m), round returns either y' or **`[`**y' **`uv-flag]`**, according to whether y is in the range of m, , n numbers and is exact at n' bits of precision. Otherwise, round returns z.

The division algorithm uses the rounder in its final step. Furthermore, our main theorem (below) claims the equality of divide(p, d, *mode*) and round(p/d, *mode*). Thus, the rounder is involved in both the implementation and specification.

We claim that round(x, [s m n uvp]) "correctly" rounds x in the specified style s and returns either the appropriate (possibly denormal) result or an appropriate indication of overflow, etc. As is the case for the other basic blocks

Algorithm divide$(p, d, mode)$						
1.	sd_0	$=$ lookup(d)	[exact	17	8]	*; 8-bit approx of 1/d*
2.	d_r	$= d$	[away	17	32]	
3.	sdd_0	$= sd_0 \times d_r$	[away	17	32]	*; first NR iteration*
4.	sd_1	$= sd_0 \times$ comp$(sdd_0, 32)$	[trunc	17	32]	
5.	sdd_1	$= sd_1 \times d_r$	[away	17	32]	*; second NR iteration*
6.	sd_2	$= sd_1 \times$ comp$(sdd_1, 32)$	[trunc	17	32]	*; 32-bit approx of 1/d*
7.	dh	$= d$	[trunc	17	32]	*; prepare for*
8.	dl	$= d - dh$	[exact	17	32]	*; quotient digit calc*
9.	p_0	$= p$	[exact	17	64]	
10.	ph_0	$= p_0$	[trunc	17	32]	
11.	q_0	$= sd_2 \times ph_0$	[away	17	24]	*; quotient digit 0*
12.	qdh_0	$= q_0 \times dh$	[exact	17	64]	
13.	qdl_0	$= q_0 \times dl$	[exact	17	64]	
14.	pt_1	$= p_0 - qdh_0$	[exact	17	64]	
15.	p_1	$= pt_1 - qdl_0$	[exact	17	64]	*; partial remainder 1*
16.	ph_1	$= p_1$	[trunc	17	32]	
17.	q_1	$= sd_2 \times ph_1$	[away	17	24]	*; quotient digit 1*
18.	qdh_1	$= q_1 \times dh$	[exact	17	64]	
19.	qdl_1	$= q_1 \times dl$	[exact	17	64]	
20.	pt_2	$= p_1 - qdh_1$	[exact	17	64]	
21.	p_2	$= pt_2 - qdl_1$	[exact	17	64]	*; partial remainder 2*
22.	ph_2	$= p_2$	[trunc	17	32]	
23.	q_2	$= sd_2 \times ph_2$	[away	17	24]	*; quotient digit 2*
24.	qdh_2	$= q_2 \times dh$	[exact	17	64]	
25.	qdl_2	$= q_2 \times dl$	[exact	17	64]	
26.	pt_3	$= p_2 - qdh_2$	[exact	17	64]	
27.	p_3	$= pt_3 - qdl_2$	[exact	17	64]	*; partial remainder 3*
28.	ph_3	$= p_3$	[trunc	17	32]	
29.	q_3	$= sd_2 \times ph_3$	[trunc	17	24]	*; quotient digit 3*
30.	qq_2	$= q_2 + q_3$	[sticky	17	64]	*; sum the digits*
31.	qq_1	$= qq_2 + q_1$	[sticky	17	64]	
32.	$divide$	$= qq_1 + q_0$	*mode*			*; (see notes)*

Fig. 1. The Division Algorithm.

(adder, multiplier, etc.), we do not descend into the block and prove that the unit functions correctly. To do so for the rounder would require formalizing independently (say, from the IEEE standard) what "correct" rounding is. See [37]. A look at our main theorem reveals that we proved the division algorithm produces the same thing as calling the rounder on p/d. But, to prove our main result, we had to prove some facts about the rounder. In addition, we proved some other facts merely to reassure ourselves of its properties. Suppose x is a rational and [*s m n uvp*] is a rounding mode.

LEMMA 2.3.1. round(x, *mode*) = round(*sticky*(x, 66), *mode*).

Note: It follows that the above two invocations of the rounder signal the same exceptions, etc., since that information is represented in the result of the function round.

LEMMA 2.3.2. *If* MinN(m) $\leq |x| \leq$ MaxN(m, n) *(i.e.,* x *is within the normal m,,n range), then* round(x, *mode*) = roundx(s, x, n), *i.e., the result is just the ideal round with the requested style s to the requested precision n.*

Note: It follows that no exception can occur since roundx is rational. The following corollary is useful.

LEMMA 2.3.3. *If x fits exactly in the normalized m,,n format, then* round(x, *mode*) = x.

LEMMA 2.3.4. *If underflows are masked and* MinD(m, n) $\leq |x| <$ MinN(m, n) *(i.e.,* x *is in the denormal part of the m,,n range), then the numeric result of the rounder is an ideal round of x at a possibly lower precision* $0 < i \leq n$.

Of course, other theorems could be proven about the rounder, e.g., that it is IEEE compliant or, at least, that it constructs numbers in the *m,,n* format. But, we did not prove those theorems for this project.

3 THE ALGORITHM

The floating-point division algorithm, called "divide" and shown in Fig. 1, takes three inputs: floating-point numbers p and d and a "rounding mode" *mode*. In the code, three styles of normalized rounding are used: **`trunc`** rounds toward 0, **`away`** rounds away from 0, and **`sticky`** rounds toward 0 if no precision is lost or the last bit kept is `1`, and rounds away from 0 otherwise.

The algorithm consists of 32 assignment statements, each of the form

i. *var = expr* *tuple* *; optional comment*

where i is a line number, *var* is a variable symbol, *expr* is a mathematical expression, and *tuple* is of the form [*style m n*] (except for the last line).

TABLE 1
SKETCH OF THE INVERSE TABLE

top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse
1.0000000_2	0.11111111_2	1.0100000_2	0.11001100_2	1.1000000_2	0.10101010_2	1.1100000_2	0.10010010_2
1.0000001_2	0.11111101_2	1.0100001_2	0.11001011_2	1.1000001_2	0.10101001_2	1.1100001_2	0.10010001_2
1.0000010_2	0.11111011_2	1.0100010_2	0.11001010_2	1.1000010_2	0.10101000_2	1.1100010_2	0.10010001_2
...	...	...	...	...	...	...	...
1.0011101_2	0.11010000_2	1.0111101_2	0.10101101_2	1.1011101_2	0.10010100_2	1.1111101_2	0.10000001_2
1.0011110_2	0.11001111_2	1.0111110_2	0.10101100_2	1.1011110_2	0.10010011_2	1.1111110_2	0.10000001_2
1.0011111_2	0.11001101_2	1.0111111_2	0.10101011_2	1.1011111_2	0.10010011_2	1.1111111_2	0.10000000_2

The lines are executed sequentially. Each line but the last either assigns a rational value to its variable or aborts the computation. To execute a line, evaluate the expression on the line to its precise mathematical value, *val*. Then, round *val* to *n* bits of precision, using *style* rounding, producing a rational, *rval*. Consider the normalized significand of *rval* and the corresponding exponent. If the exponent does not fit in *m* bits (biased representation), abort. Otherwise, assign *rval* to the variable and proceed. Lines containing a *tuple* of the form `[exact m n]` are handled slightly differently. Such a line assigns *var* the value, *val*, of the expression, but aborts unless the normalized *val* fits in the *m*,,*n* format.

We prove that no line aborts, i.e., all normalized results fit in the described registers. These claims hold when the input *p* and *d* are 15,,64 numbers and $d \neq 0$.

The last line assigns to *divide* the result of rounding $qq_1 + q_0$ according to the user-specified *mode*. This is the only occasion in this code in which a denormal number may be produced. It is also the only line that might signal an exception. Formalizing this step in ACL2 required modeling some aspects of the AMD5$_K$86's rounder. We modeled its sensitivity to an indicated rounding style and destination format, made provisions for signaling overflows and underflows, the (possibly denormal) floating-point result returned (if any), and whether the underflow flag is set.

Lines 1 through 6 of Fig. 1 are devoted to the computation of the reciprocal of *d*. At line 6, the variable sd_2 is assigned a 17,,32 normalized floating-point number that (we will prove) is $1/d$ with a relative error less than 2^{-28}. This is done by obtaining an initial approximation, sd_0, via the function lookup, which maps a 17,,64 normalized floating-point number *d* into an approximation of $1/d$ by using a table that maps each of the 128 8-bit nonzero significands to an 8-bit approximation of its reciprocal. See Table 1. The computation of the table entries is discussed in [34].

The initial approximation is then refined with two iterations of an easily computed variation of the Newton-Raphson method,

$$sd_{i+1} = sd_i(2 - sd_i \times d) \quad (0 \leq i \leq 1).$$

The variation is obtained by making the following transformations on the equation above.

- Instead of *d*, we use the floating-point number obtained by rounding *d* with the tuple `[away 17 32]` so we can use a small square multiplier in the next step.
- After multiplying sd_i by (the approximation to) *d*, we truncate the result to 32 bits. This is our sdd_i and can be thought of as an approximation to $sd_i \times d$.
- Instead of subtracting the result from 2 to form $(2 - sd_i \times d)$, we complement sdd_i, which yields a close approximation, where comp(*x*, *i*) is defined as (trunc(2 – x – 2^{1-i}, i).
- After multiplying by sd_i, we truncate the result to 32 bits.

Lines 7 through 29 of the algorithm are devoted to the computation of four quotient digits, q_0 through q_3. Each quotient digit is a 17,,24 normalized floating-point number. See lines 11, 17, 23, and 29.

Successive partial remainders are defined with the equation $p_{i+1} = p_i - (q_i \times d)$. However, computing the right-hand side directly would require a 24×64-bit multiply. We do the multiplication by splitting *d* into two 32 bit parts, d_h and d_l, then subtracting each product from p_i. See lines 12-15 for the computation of p_1. The 17,,64 normalized floating-point result is exact; no precision is lost and p_{i+1} as computed is indeed the mathematical $p_i - (q_i \times d)$.

In lines 30 through 32, the quotient digits are summed, least significant digits first, using sticky rounding on all but the last. On line 32, the final sum is rounded into the destination format via the user-specified *mode*.

4 THE MAIN THEOREM

MAIN THEOREM. *If p and d are* 15,,64 *floating-point numbers,* $d \neq 0$, *and mode is a rounding mode, then* divide(*p*, *d*, *mode*) = round(p/d, *mode*).

In Section 9, we list some corollaries of this theorem, based on Lemmas 2.3.2-2.3.4.

5 THE MAIN PROOF

To facilitate discussion about the values of the variables in the code, we define auxiliary functions corresponding to the 32 variable names. Consider, for example, line 6.

6. $sd_2 = sd_1 \times \text{comp}(sdd_1, 32)$ `[trunc 17 32]`

The semantics of this line can be rendered as the following function of *d*, provided we have rendered the preceding lines as analogous functions:

$$\text{esd}_2(d) = \text{eround}(\text{esd}_1(d) \times \text{comp}(\text{esdd}_1(d), 32), [\texttt{trunc 17 32}])$$

where eround(*val*, `[s m n]`) either returns roundx(*s*, *val*, *n*),

val (when *s* is **exact**), or signals an "abort," as described in Section 3.

We call esd_2 the *semantic function* for the code variable "sd_2." Assuming no previous line signals an error, the meaning of the code variable "sd_2" is $\text{esd}_2(d)$. More generally, if the division algorithm is executed with $d = x$ and executes to line 6 with no error, then $\text{esd}_2(x)$ is the value of "sd_2."

Because we always assume no previous line has signaled an error when we use $\text{esd}_2(d)$, we need not concern ourselves with the meaning of the definition above when $\text{esdd}_1(d)$, say, is an error object. But, $\text{esd}_2(d)$ may itself signal the first error. Thus, it is somewhat awkward to reason about $\text{esd}_2(d)$ because it is not always numeric.

Now, consider the function

$$\text{sd}_2(d) = \text{trunc}(\text{sd}_1(d) \times \text{comp}(\text{sdd}_1(d), 32), 32).$$

When the variable sd_2 in the code has a numeric value—as opposed to an error object value—that value is given by our function sd_2. In particular, $\text{esd}_2(d) = \text{sd}_2(d)$ unless the former is an error object. Furthermore, the former is an error object if and only if the precision or exponent magnitude of the latter is too big. Thus, not only are the "numeric semantic functions" like sd_2 easier to handle, but their arithmetic properties can be readily traded in for properties about the true semantics, e.g., esd_2. The numeric semantic functions for lines containing **[exact** *m n***]** assertions return the value of the expression, e.g., the numeric semantic function for line 15 is $\text{p}_1(p, d) = \text{pt}_1(p, d) - \text{qdl}_0(p, d)$. The numeric semantic function for the last line, named divide! to distinguish it from divide, is divide!(p, d, *mode*) = round($\text{qq}_1(p, d) + \text{q}_0(p, d)$, *mode*). We call divide! the *numeric version* of the algorithm, although it is not necessarily numerically valued, since the rounder can indicate exceptions, etc.

Henceforth, when we use a code variable (other than p, d, and *mode*), it is merely an abbreviation for the application of the corresponding numeric semantic function to the appropriate subsequence of p, d, and *mode*. For example, consider line 11. Henceforth, q_0 is just an abbreviation for $\text{q}_0(p, d)$, which, using the same convention, is away($sd_2 \times ph_0$, 24) or away($\text{sd}_2(d) \times \text{ph}_0(p)$, 24).

We will prove that the numeric algorithm enjoys an even stronger correctness property than the actual algorithm:

THEOREM 1. *If p and d are rational numbers, $d \neq 0$, and mode is a rounding mode, then* divide!(p, d, *mode*) = round(p/d, *mode*).

The 15,,64 hypotheses are necessary only to relate the numeric algorithm to the actual one. We will also prove:

THEOREM 2. *If p and d are* 15,,64 *floating-point numbers, $d \neq 0$, and mode is a rounding mode, then* divide!(p, d, *mode*) = divide(p, d, *mode*).

Theorems 1 and 2 together imply the main theorem. We prove Theorem 1 in Section 7 and Theorem 2 in Section 8.

6 Fundamental Elementary Results

In this section, we list a few of the lemmas that we had to formalize and prove in order to construct our main proofs. The library of such lemmas contains hundreds of entries. We show only a few here to give the reader a sense of where our proof starts.

In the following lemmas, x and y are rationals and i is a positive integer.

6.1 Key Properties of the Representation

LEMMA 6.1.1. $\text{s}(-x) = \text{s}_x$ *and* $\text{e}(-x) = \text{e}_x$.

LEMMA 6.1.2. *If j is an integer,* $\text{s}(x \times 2^j) = \text{s}_x$ *and* $\text{e}(x \times 2^j) = \text{e}_x + j$ $(x \neq 0)$.

LEMMA 6.1.3. *If $x \neq 0$ and j is an integer and $|x| < 2^j$, then $\text{e}_x < j$.*

LEMMA 6.1.4. *If $x \neq 0$ and j is an integer and $2^j \leq |x|$, then $j \leq \text{e}_x$.*

An upper bound on the exponent of a sum (or difference) is given by:

LEMMA 6.1.5. *If $x \neq 0$ and $x + y \neq 0$ and $\text{e}_y \leq \text{e}_x$, then $\text{e}(x + y) \leq 1 + \text{e}_x$.*

A lower bound on the exponent of a sum (or difference) of two numbers (when the exponent of one is sufficiently smaller than that of the other) is given by

LEMMA 6.1.6. *If $x \neq 0$ and $y \neq 0$ and $\text{e}_y + 1 < \text{e}_x$, then $\text{e}_y < \text{e}(x + y)$.*

Bounds on the exponent of a product are given by:

LEMMA 6.1.7. *If $x \neq 0$ and $y \neq 0$, then $\text{e}_x + \text{e}_y \leq \text{e}(x \times y) \leq \text{e}_x + \text{e}_y + 1$.*

6.2 Elementary Properties of Rounding

For most of the lemmas below about trunc we also proved analogous versions about away and sticky.

LEMMA 6.2.1. $\text{trunc}(-x, i) = -\text{trunc}(x, i)$.

LEMMA 6.2.2. *For every x, there is an ε of the same sign such that* $\text{trunc}(x, i) = x - \varepsilon$ *and* $|\varepsilon| < 2^{\text{e}_x - i + 1}$.

By "same sign" here we mean that if $x < 0$, then $\varepsilon \leq 0$, and, otherwise, $0 \leq \varepsilon$.

LEMMA 6.2.3. *If $x \leq y$, then* $\text{trunc}(x, i) \leq \text{trunc}(y, i)$.

LEMMA 6.2.4. *If $x \neq 0$, then* $\text{e}(\textit{trunc}(x, i)) = \text{e}_x$.

LEMMA 6.2.5. *If $x \neq 0$, then*

$$\text{e}\big(\textit{away}(x, i)\big) = \begin{cases} 1 + \text{e}_x & \text{if } 2 < s_x + 2^{-i+1} \\ \text{e}_x & \text{otherwise.} \end{cases}$$

LEMMA 6.2.6. *If j is an integer, then* $\text{trunc}(x \times 2^j, i) = \text{trunc}(x, i) \times 2^j$.

LEMMA 6.2.7. *If j is an integer and $i \leq j$, then* $\text{trunc}(\text{trunc}(x, i), j) = \text{trunc}(x, i)$.

LEMMA 6.2.8. *If j is an integer and $i \leq j$, then* $\text{trunc}(\text{away}(x, i), j) = \text{away}(x, i)$.

Since we use the test "$\text{trunc}(x, n) = x$" to formalize "x fits in n bits," it is Lemma 6.2.8 that captures the remark that "the result of rounding away to i bits fits in j bits if $i \leq j$." Similar results hold for the other rounding styles.

The following lemmas about trunc are useful in exactness arguments.

LEMMA 6.2.9. *If m and n are positive integers and* $\text{trunc}(x, m) = x$ *and* $\text{trunc}(y, n) = y$, *then* $\text{trunc}(x \times y, m + n) = x \times y$.

LEMMA 6.2.10. *For every rational x and positive integer i, there is an integer j such that* $\text{trunc}(x, i) = j \times 2^{\text{e}_x - i + 1}$, *where* $\sigma_j = \sigma_x$ *and* $2^{i-1} \leq |j| < 2^i$.

LEMMA 6.2.11. *If n is an integer and $|n| < 2^i$, then* $\text{trunc}(n, i) = n$.

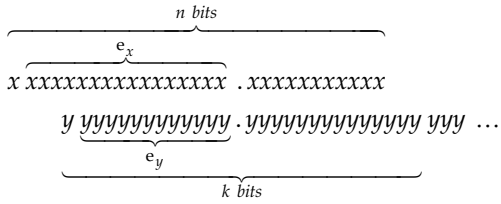
6.3 Special Properties of Sticky Rounding

We are especially interested in the interactions between sticky rounding and the other styles because of its crucial use to sum the quotient digits.

LEMMA 6.3.1. *If s is one of the six rounding styles and i and n are positive integers such that* $i \le n$, *then* roundx(s, sticky(x, $n+2$), i) = roundx(s, x, i).

An especially important fundamental result is

LEMMA 6.3.2 (Sticky Plus). *Let x be a nonzero rational that fits in* $n > 0$ *bits, which is to say* trunc(x, n) = x. *Let y be a rational whose exponent is at least two smaller than that of x,* $1 + e_y < e_x$. *Let k be a positive integer such that* $n + e_y - e_x < k$.



Then, sticky($x + y$, n) = sticky(x + sticky(y, k), n).

The need for Lemma 6.3.2, which is henceforth called "Sticky Plus," can be informally explained as follows: Suppose one wishes to round the "infinitely precise" sum $x + y$ to n bits with sticky rounding but one only has a finite number of bits in which to compute the sum. Suppose x itself fits in n bits but y is "infinitely precise" and is as described by the lemma above. Then, one can first sticky round the "infinitely precise" y to k bits, do a finite sum, and sticky round the result to obtain the desired answer. This is the property of sticky rounding that allows us to sum the quotient digits without endangering the round of the infinitely precise answer.

Among the lemmas noted in this section, we found Sticky Plus to be singularly difficult to prove. Below, we sketch a proof due to Russinoff, who checked his proof with ACL2 during his analysis of the AMD5$_K$86 floating-point square root microcode [42]. Russinoff's Sticky Plus proof is more elegant than the one we checked.

Instead of proving the lemma for all k, Russinoff proves the special case where $k = n - e_x + e_y + 1$, the least k satisfying of our hypotheses; the extension to greater k is straightforward using a lemma for sticky similar to our Lemma 6.2.7. For brevity, we concern ourselves primarily with the "positive case" (x and y both positive).

We start with three observations. First, x fits in n bits iff $2^{n-1-e_x}x$ is an integer. Second, sticky(x, n) is either trunc(x, n) or away(x, n) according to whether $\left\lfloor 2^{n-1-e_x}x \right\rfloor$ is odd or even. Third, if x fits in n bits, where $n = k + e_x - e_y > 0$, then

1) x + trunc(y, k) = trunc($x + y$, $k + e(x + y) - e_y$) and
2) x + away(y, k) = away($x + y$, $k + e(x + y) - e_y$).

Russinoff proves 1) in [42]; the proof of 2) is similar.

The positive case of Sticky Plus can then be proven from the following lemma: If x fits in $n = k - 1 + e_x - e_y > 0$ bits, then x + sticky(y, k) = sticky($x + y$, $k + e(x + y) - e_y$).

PROOF. Note that $2^{k-2-e_y}x = 2^{n-1-e_x}x$ is an integer. By the third observation above, it suffices to show that

$$\left\lfloor 2^{k-1-e_y}y \right\rfloor \equiv \left\lfloor 2^{k+e(x+y)-e_y-1-e(x+y)}(x+y) \right\rfloor \pmod 2.$$

But,

$$\begin{aligned}\left\lfloor 2^{k+e(x+y)-e_y-1-e(x+y)}(x+y) \right\rfloor &= \left\lfloor 2^{k-1-e_y}(x+y) \right\rfloor \\ &= \left\lfloor 2\times 2^{k-2-e_y}x + 2^{k-1-e_y}y \right\rfloor \\ &= 2\times 2^{k-2-e_y}x + \left\lfloor 2^{k-1-e_y}y \right\rfloor \\ &\equiv \left\lfloor 2^{k-1-e_y}y \right\rfloor \pmod 2.\end{aligned}$$

□

The negative case of Sticky Plus is analogous but relies on the fact that $\lfloor x \rfloor = -\lceil -x \rceil$ and a swapped version of our third observation above, e.g., x − trunc(y, k) = away($x - y$, k) + $e(x - y) - e_y$). In the negative case, we must prove that the parities are essentially opposite.

We conclude this section with one more important lemma about sticky rounding.

LEMMA 6.3.3. *Let x be a nonzero rational such that* trunc(x, n) = x, *where* $n > 1$. *Let* ε_1 *and* ε_2 *be nonzero rationals such that* $|\varepsilon_1| < 2^{e_x - n + 1}$ *and* $|\varepsilon_2| < 2^{e_x - n + 1}$. *Furthermore, suppose both* ε_1 *and* ε_2 *are positive if either is (i.e.,* $0 < \varepsilon_1 \leftrightarrow 0 < \varepsilon_2$). *Then,* sticky($x + \varepsilon_1$, n) = sticky($x + \varepsilon_2$, n).

7 PROOF OF THEOREM 1

The following proof is a "journal level" description of the one we checked with ACL2. Recall that code variables here denote calls of the numeric semantic functions.

THEOREM 1. *If p and d are rational numbers,* $d \neq 0$, *and mode is a rounding mode, then* divide!(p, d, *mode*) = round(p/d, *mode*).

PROOF. Assume p and d are rationals and $d \neq 0$.

The first six lines of the numeric algorithm compute an approximation to the reciprocal of d. In Subsection 7.1, we will prove

LEMMA 7.1.1. *For every nonzero rational d, there exists a rational* $0 \le \varepsilon_{sd2} < 2^{-28}$ *such that* $sd_2 = (1/d)(1 - \varepsilon_{sd2})$.

This lemma will enable us to prove the crucial properties of the quotient digits, namely, that their exponents differ by at least 23. The crucial lemma relating q_0 to q_1 for example is

LEMMA 7.2.1 (Digit Separation (q_0 *v.* q_1)). *If p and d are rationals,* $d \neq 0$, *and* $q_1 \neq 0$, *then* $e(q_1) \le e(q_0) - 23$.

The Digit Separation lemma (Section 7.2) states an analogous or slightly stronger property for all three quotient digits, as well as for what we will call q_3' below.

One implication of Digit Separation is that two nonzero quotient digits have a nonzero sum. For example, if q_2 and q_3 are nonzero, then $q_2 + q_3$ is nonzero, for, otherwise, the exponents of q_2 and q_3 would be

equal, since $e(-q_3) = e(q_3)$. We use these and similar observations implicitly below.

Lines 7 through 9 of the code prepare for the quotient digit calculation by defining dh and dl to be the high and low parts, respectively, of d, and renaming p to be p_0 so the subsequent indexing is regular. Hence, $dh + dl = d$. Note that, in the actual algorithm (as opposed to the numeric one we are discussing), "dl" is $d - dh$ only if that quantity fits exactly in 32 bits or less. We deal with this, of course, when we work on Theorem 2.

The first quotient digit, q_0, and the next partial remainder, p_1, are computed by lines 10 through 15. Unwinding the definition of p_1 gives $p_1 = pt_1 - qdl_0 = (p_0 - qdh_0) - qdl_0 = p_0 - (qdh_0 + qdl_0) = p_0 - (q_0 \times dh + q_0 \times dl) = p_0 - q_0 \times (dh + dl) = p_0 - q_0 \times d$.

The computation of the next two quotient digits and remainders is analogous. Thus, unwinding as above, we get $p_3 = p_0 - (q_0 + q_1 + q_2) \times d$. If we define q_3' to be p_3/d, it follows that $p_0 = (q_0 + q_1 + q_2 + q_3') \times d$, which is to say

$$p/d = p_0/d = (q_0 + q_1 + q_2 + q_3'). \quad (1)$$

Equation (1) states that the "infinitely precise" answer is the sum of the first three quotient digits plus q_3'. Note, however, that the algorithm does not compute q_3' but $q_3 = \text{trunc}(sd_2 \times \text{trunc}(p_3, 32), 24)$, which is generally different.

In this paper, we address only the case where all four quotient digits are nonzero and leave the other cases to the reader. Hint: If one quotient digit is 0, all subsequent ones are 0.

The final steps of the computation sum the quotient digits.

$$divide! = \text{round}(q_0 + \text{sticky}(q_1 + \text{sticky}(q_2 + q_3, 64), 64), mode).$$

But, the rounder is only sensitive to 66 bits (Lemma 2.3.1), so this is equivalent to

$$divide! = \text{round}(\psi, mode), \quad (2)$$

where

$$\psi = \text{sticky}(q_0 + \text{sticky}(q_1 + \text{sticky}(q_2 + q_3, 64), 64), 66). \quad (3)$$

We will show that

$$\psi = \text{sticky}(q_0 + \text{sticky}(q_1 + \text{sticky}(q_2 + \text{sticky}(q_3, 2), 24), 45), 66). \quad (4)$$

To prove this, we reduce the right-hand sides of both (3) and (4) to $\text{sticky}(q_0 + q_1 + q_2 + q_3, 66)$.

The reduction of (4) repeatedly applies Sticky Plus, starting on the inside and working out, appealing to Digit Separation and our nonzero sum observations to relieve the hypotheses:

$$\begin{aligned}\psi &= \text{sticky}\big(q_0 + \text{sticky}\big(q_1 + \text{sticky}\big(q_2 + \text{sticky}(q_3, 2), 24\big), 45\big), 66\big)\\ &= \text{sticky}\big(q_0 + \text{sticky}\big(q_1 + \text{sticky}\big(q_2 + q_3, 24\big), 45\big), 66\big)\\ &= \text{sticky}\big(q_0 + \text{sticky}\big(q_1 + q_2 + q_3, 45\big), 66\big)\\ &= \text{sticky}\big(q_0 + q_1 + q_2 + q_3, 66\big).\end{aligned}$$

Following the same procedure, we reduce the definition of ψ, (3), to the same term and, hence, have proven (4).

But, we can replace q_3 in the right-hand side of (4) by q_3' to get

$$\begin{aligned}\psi &= \text{sticky}\big(q_0 + \text{sticky}\big(q_1 + \text{sticky}\big(q_2 + \text{sticky}(q_3, 2), 24\big), 45\big), 66\big)\\ &= \text{sticky}\big(q_0 + \text{sticky}\big(q_1 + \text{sticky}\big(q_2 + \text{sticky}(q_3', 2), 24\big), 45\big), 66\big) \quad (5)\end{aligned}$$

This is justified because $\text{sticky}(q_3, 2)$ and $\text{sticky}(q_3', 2)$ satisfy the hypotheses on ε_1 and ε_2 of Lemma 6.3.3. In particular, Digit Separation implies

$$0 < \big|\text{sticky}(q_3, 2)\big| < 2^{e(q_2)-23}$$

and

$$0 < \big|\text{sticky}(q_3', 2)\big| < 2^{e(q_2)-23}.$$

It is also true that $0 < \text{sticky}(q_3, 2)$ if and only if $0 < \text{sticky}(q_3', 2)$.

Now, we eliminate the inner sticky terms from the right-hand side of (5) with Sticky Plus and Digit Separation (including the one for q_3'), just as we did when we proved (4) above:

$$\psi = \text{sticky}\big(q_0 + q_1 + q_2 + q_3', 66\big). \quad (6)$$

Thus, we have

$$\begin{aligned}&divide!\\ &= \text{round}(\psi, mode) && \text{by (2)}\\ &= \text{round}\big(\text{sticky}\big(q_0 + q_1 + q_2 + q_3', 66\big), mode\big) && \text{by (6)}\\ &= \text{round}\big(\text{sticky}\big(p/d, 66\big), mode\big) && \text{by (1)}\\ &= \text{round}\big(p/d, mode\big) && \text{by (Lemma 2.3.1).}\end{aligned}$$

□

7.1 The Reciprocal Computation

LEMMA 7.1.1. *For every nonzero rational d there exists a rational $0 \le e_{sd2} < 2^{-28}$ such that $sd_2 = (1/d)(1 - e_{sd2})$.*

To give the reader a feel for the mechanization of such proofs, we describe this one at a fairly low level. Please refer to lines 1 through 6.

Our proof of Lemma 7.1.1 is based on the observation that, without loss of generality, we can restrict our attention to the case where $1 \le d < 2$. We call this a "squeeze" lemma. To make this formal, we first observe

LEMMA 7.1.2. *If d is a rational and $d \neq 0$, then $\text{sd}_2(d) = \sigma(d) \times \text{sd}_2(s(d)) \times 2^{-e(d)}$.*

```
(DEFTHM SD2-SQUEEZE
  (IMPLIES (AND (RATIONALP D)
                (NOT (EQUAL D 0)))
           (EQUAL (SD2 D)
                  (* (SIGN D)
                     (SD2 (SIGNIFICAND D))
                     (EXPT 2 (- (EXPO D)))))))
```

Fig. 2. Lemma 7.1.2.

```
Subgoal 1
(IMPLIES (AND (RATIONALP D)
              (NOT (EQUAL D 0))
              (<= 0 D))
         (EQUAL (TRUNC (* (SD1 (SIGNIFICAND D))
                          1 (EXPT 2 (* -1 (EXPO D)))
                          (COMP (SDD1 (SIGNIFICAND D)) 32))
                       32)
                (* 1 (EXPT 2 (* -1 (EXPO D)))
                   (TRUNC (* (SD1 (SIGNIFICAND D))
                             (COMP (SDD1 (SIGNIFICAND D)) 32))
                          32)))).

By the simple :rewrite rule UNICITY-OF-1 we reduce the conjecture to

Subgoal 1'
(IMPLIES (AND (RATIONALP D)
              (NOT (EQUAL D 0))
              (<= 0 D))
         (EQUAL (TRUNC (* (SD1 (SIGNIFICAND D))
                          (FIX (* (EXPT 2 (* -1 (EXPO D)))
                                  (COMP (SDD1 (SIGNIFICAND D)) 32))))
                       32)
                (FIX (* (EXPT 2 (* -1 (EXPO D)))
                        (TRUNC (* (SD1 (SIGNIFICAND D))
                                  (COMP (SDD1 (SIGNIFICAND D)) 32))
                               32))))).

But simplification reduces this to T, using the :type-prescription rules TRUNC, EXPO,
SDD1, SD1, RATIONALP-COMP, RATIONALP-EXPT and POSITIVE-EXPT-2-I, the :definition FIX,
the :rewrite rules TRUNC-*-X-EXPO-BRIDGE-1 and COMMUTATIVITY-OF-* and primitive type
reasoning.
```

Fig.3 . Proof of the first subgoal.

PROOF.

$$\mathrm{sd}_0(d) = \sigma(d) \times \mathrm{sd}_0(\mathrm{s}(d)) \times 2^{-\mathrm{e}(d)}$$
$$\mathrm{d}_r(d) = \sigma(d) \times \mathrm{d}_r(\mathrm{s}(d)) \times 2^{\mathrm{e}(d)}$$
$$\mathrm{sdd}_0(d) = \mathrm{sdd}_0(\mathrm{s}(d))$$
$$\mathrm{sd}_1(d) = \sigma(d) \times \mathrm{sd}_1(\mathrm{s}(d)) \times 2^{-\mathrm{e}(d)}$$
$$\mathrm{sdd}_1(d) = \mathrm{sdd}_1(\mathrm{s}(d))$$
$$\mathrm{sd}_2(d) = \sigma(d) \times \mathrm{sd}_2(\mathrm{s}(d)) \times 2^{-\mathrm{e}(d)}.$$

□

The mechanically checked version of this proof required first proving a squeeze lemma for each of the preceding lines. The ACL2 statement of Lemma 7.1.2 is shown in Fig. 2.

The mechanically produced proof description begins as follows:

This simplifies, using the :definitions SD2, SIGN and SYNTAXP, the :executable-counter-part of EQUAL and the :rewrite rules COMMUTATIVITY-OF-*, SDD1-SQUEEZE, SD1-SQUEEZE, A9 and A2, to the following two conjectures.

The proof of the first subgoal is shown in Fig. 3.

The other subgoal is similar. The machine's case split on the sign of `D` is unnecessary. The proof is printed as it is produced. About one-fifth of a second elapses from the time the conjecture is posed to the time the proof is complete. However, as is obvious from the lemma names cited (e.g., **SDD1-SQUEEZE**), the user "laid the groundwork" for this proof by proving the squeeze lemmas for the preceding lines. That sequence of lemmas is essentially the proof we gave above. In addition, the user had already proven rules for algebraic simplification (e.g., that 2^i is positive for all integers i). Lemmas are most often used as rewrite rules. With the appropriate choice of lemmas, the user can

TABLE 2
ERROR ANALYSIS FOR LINES 1-6 ($\delta = 2^{-29} + 2^{-31} + (9/512)2^{-31}$)

var = value	error bounds
$sd_0 = (1/d)(1+\varepsilon_{sd0}(d))$	$\lvert\varepsilon_{sd0}(d)\rvert < 2^{-8}+2^{-9}$
$sdd_0 = 1+\varepsilon_{sdd0}(d)$	$\varepsilon_{sd0}(d) \le \varepsilon_{sdd0}(d) \le \varepsilon_{sd0}(d)+2^{-30}$
$sd_1 = (1/d)(1-\varepsilon_{sd1}(d))$	$0 \le \varepsilon_{sd1}(d) \le \varepsilon_{sd0}(d)^2+\delta$
$sdd_1 = (1-\varepsilon_{sdd1}(d))$	$\varepsilon_{sd1}(d)-2^{-30} \le \varepsilon_{sdd1}(d) \le \varepsilon_{sd1}(d)$
$sd_2 = (1/d)(1-\varepsilon_{sd2}(d))$	$0 \le \varepsilon_{sd2}(d) \le \varepsilon_{sd1}(d)^2+\delta$

program ACL2 to do simplification and other symbolic manipulation, leading the system to harder proofs. See [33].

Given Lemma 7.1.2, it is easy to prove that $sd_2(s(d))$ approximates $1/s(d)$ with the same relative error that $sd_2(d)$ approximates $1/d$.

LEMMA 7.1.3. *If* $d \neq 0$ *and* $sd_2(s(d)) = (1/s(d))(1-\varepsilon)$, *then* $sd_2(d) = (1/d)(1-\varepsilon)$.

Hence, we can prove Lemma 7.1.1 by instantiation of Lemma 7.1.4, below: Replace d by $s(d)$; appeal to the fact that, for $d \neq 0$, $1 \le s(d) < 2$; and use Lemma 7.1.3.

LEMMA 7.1.4. *For every rational* d, $1 \le d < 2$, *there exists a rational* $0 \le \varepsilon_{sd2} < 2^{-28}$ *such that* $sd_2 = (1/d)(1-\varepsilon_{sd2})$.

PROOF. Suppose $1 \le d < 2$.

It is helpful to generalize away from the particulars of Table 1. Therefore, consider any table mapping keys to values. We say a table entry, $\langle k, v\rangle$ mapping key k to value v is ε-*ok* if and only if k and v are rational numbers, $0 < v$, $|kv-1| < \varepsilon$, and $|(k+2^{-7})v-1| < \varepsilon$. If we think of v as an approximation of the inverse of x for x in the range $k \le x < k+2^{-7}$, then the ε-ok condition limits the relative error at the endpoints. We say a table is ε-*ok* if every entry in it is ε-ok.

If $\langle k, v\rangle$ is ε-ok, where k is the truncation of d to 8 bits, truncn(d, 8), then it follows from the monotonicity of multiplication and $k \le d < k+2^{-7}$ that $|dv-1| < \varepsilon$. Thus, if a table is ε-ok and it contains a value v for truncn(d, 8), then $|dv-1| < \varepsilon$.

It is easy to confirm by computation that Table 1 is ε-ok for $\varepsilon = 3/512$ and that it contains an entry assigning a value for the 8-bit truncation of every $1 \le d < 2$ (e.g., the 128 8-bit nonzero significands). Hence, by the definition of lookup and the ε-ok property of the table, $|d \times \text{lookup}(d) - 1| < 3/512$.

It is convenient to define $\varepsilon_{sd0}(d)$ to be $d \times \text{lookup}(d) - 1$. It follows that $sd_0 = \text{lookup}(d) = (1/d)(1+\varepsilon_{sd0}(d))$, where $|\varepsilon_{sd0}(d)| < 3/512 = 2^{-8}+2^{-9}$.

We now move on to lines 2 through 6 of the code. Observe that if $0 \le x < 2$, then $\text{trunc}(x, 32) = x(1-\tau_x)$ for some $0 \le \tau_x < 2^{-31}$, and $\text{away}(x, 32) = x(1+\alpha_x)$ for some $0 \le \alpha_x < 2^{-31}$. These two observations, along with the definition of comp and appropriate definitions of ε_{sdd0}, ε_{sd1}, ε_{sdd1}, and ε_{sd2} (as functions of d analogous to ε_{sd0} above) allow us to derive the equations and inequalities of Table 2. From these inequalities, it readily follows that $0 \le \varepsilon_{sd2}(d) < 2^{-28}$ and, hence, Lemma 7.1.4 and, hence, Lemma 7.1.1 have both been proven. □

Perhaps the most interesting aspect of checking this proof mechanically is the ε-ok property of Table 1. Just as described above, we defined this property as an ACL2 (Common Lisp) predicate, `table-okp`, and proved the general lemma stating that any table satisfying that predicate gives sufficiently accurate answers. When the general lemma is applied to our particular lookup, the system must prove `(table-okp *divide-table* 3/512)`, where `*divide-table*` is a list corresponding to Table 1. This proof is by evaluation, since no variables are involved. The computation takes about 0.01 seconds. Thus, the only time the particulars of Table 1 are involved in the proof is when the predicate is executed. This example illustrates the value of computation in a general-purpose logic.

7.2 Digit Separation

LEMMA 7.2.1 (Digit Separation). *Suppose that* p *and* d *are rationals and* $d \neq 0$. *Let* $q_3' = p_3/d$. *Then,*

$$\begin{aligned} q_1 \neq 0 &\rightarrow e(q_1) \le e(q_0) - 23,\\ q_2 \neq 0 &\rightarrow e(q_2) \le e(q_1) - 23,\\ q_3 \neq 0 &\rightarrow e(q_3) < e(q_2) - 23, \text{ and}\\ q_3' \neq 0 &\rightarrow e(q_3') < e(q_2) - 23. \end{aligned}$$

PROOF. In this paper, we will prove only the first of the four implications above, namely $q_1 \neq 0 \rightarrow e(q_1) \le e(q_0) - 23$. The others are analogous. The relevant lines of code for the first implication are lines 10 through 17.

Assume p and d are rationals, $d \neq 0$, and $q_1 \neq 0$. The desired conclusion,

$$e(q_1) \le e(q_0) - 23$$

is equivalent to

$$e(\text{away}(sd_2 \times ph_1, 24)) \le e(q_0) - 23. \tag{7}$$

By fundamental theorems about e, trunc, and away, (7) is implied by $|sd_2 \times ph_1| < 2^{e(q_0)-23}$, which is equivalent to $|sd_2| \times |\text{trunc}(p_0 - d \times q_0, 32)| < 2^{e(q_0)-23}$, which is, in turn, implied by

$$|sd_2| \times |p_0 - d \times q_0| < 2^{e(q_0)-23}. \tag{8}$$

In perhaps the most surprising move of the proof, we now rewrite the left hand side above to express (8) equivalently as

$$\begin{aligned} &|sd_2(p_0 - sd_2 \times d \times ph_0)\\ &\quad + sd_2 \times d(sd_2 \times ph_0 - q_0)| < 2^{e(q_0)-23}. \end{aligned} \tag{9}$$

Let $\alpha = sd_2(p_0 - sd_2 \times d \times ph_0)$ and $\beta = sd_2 \times d\ (sd_2 \times ph_0 - q_0)$. Then, (9) has the form

$$|\alpha + \beta| < 2^{e(q_0)-23}. \quad (10)$$

But, as we will show, α and β have different signs and their absolute values are bounded strictly above by $2^{e(q_0)-23}$. But, in this case, it follows that (10) is true.

We first show that α and β have different signs. Then we bound each of them.

By "different signs" here we mean that one is nonpositive and the other is nonnegative, i.e., $((\alpha \le 0 \wedge 0 \le \beta) \vee (\beta \le 0 \wedge 0 \le \alpha))$. First, observe that, since α and β share a factor of sd_2, we can cancel. Simple arithmetic, therefore, gives us that α and β have different signs if and only if $(p_0 - sd_2 \times d \times ph_0)$ and $(sd_2 \times d \times ph_0 - d \times q_0)$ have different signs. Now, note that the two expressions whose signs we are comparing are of the form $x - y$ and $y - z$, where x is p_0, y is $sd_2 \times d \times ph_0$, and z is $d \times q_0$.

The following easily proved arithmetic lemma allows us to reduce the question to this lemma's Conditions 1-4.

LEMMA 7.2.2. *If x, y, and z are rationals, then $x - y$ and $y - z$ have different signs if*

1) $|y| \le |x|$ *and*
2) $|y| \le |z|$ *and either*
3) $0 < x \wedge 0 \le y \wedge 0 \le z$ *or*
4) $x \le 0 \wedge y \le 0 \wedge z \le 0$.

Under the instantiation of x, y, and z above, Condition 1 becomes $|sd_2 \times d \times ph_0| \le |p_0|$. But, we know $0 < sd_2 \times d \le 1$ by Lemma 7.1.1, which tells us that sd_2 approximates $1/d$ from below. Since $|ph_0| = |\text{trunc}(p_0, 32)| \le |p_0|$, Condition 1 is proven.

Condition 2 becomes $|sd_2 \times d \times ph_0| \le |d \times q_0|$. Canceling $|d|$ and expanding the definition of q_0 gives $|sd_2 \times ph_0| \le |\text{away}(sd_2 \times ph_0, 32)|$, which proves Condition 2.

Finally, we must show either Condition 3 or Condition 4, which just split on whether x is positive. Here, we handle only the case that $0 < x$, i.e., Condition 3. We must, therefore, show $0 \le sd_2 \times d \times ph_0$ and $0 \le d \times q_0$, given $0 < p_0$. But, $sd_2 \times d$ is always positive and $0 \le ph_0$ when $0 \le p_0$. Thus, the first conjunct is true. As for the second, $d \times q_0$ is $d \times \text{away}(sd_2 \times ph_0, 32)$ which is positive if p_0 is. Thus, the second conjunct is true.

This completes the argument that α and β have different signs. We now turn to the question of bounding them. We wish to show that $|\alpha| < 2^{e(q_0)-23}$ and $|\beta| < 2^{e(q_0)-23}$. We address the second first because it is simpler.

Recalling the definition of β from earlier in this section, we wish to prove

$$\left|sd_2 \times d\left(sd_2 \times ph_0 - q_0\right)\right| < 2^{e(q_0)-23}.$$

Since $0 < sd_2 \times d \le 1$, it suffices to show

$$\left|sd_2 \times ph_0 - q_0\right| < 2^{e(q_0)-23}. \quad (11)$$

Expanding the definition of q_0 gives

$$\left|sd_2 \times ph_0 - \text{away}\left(sd_2 \times ph_0,\ 24\right)\right| < 2^{e(q_0)-23}.$$

But, this follows from $|x - \text{away}(x, i)| < 2^{e(away(x,i))-i+1}$, which is easily proven from the away-analogue of Lemma 6.2.2 together with Lemma 6.2.5.

So, now we turn to the α bound. We wish to prove $\left|sd_2(p_0 - sd_2 \times d \times ph_0)\right| < 2^{e(q_0)-23}$. We will prove the stronger $\left|sd_2(p_0 - sd_2 \times d \times ph_0)\right| < 2^{e(q_0)-24}$. Since $|sd_2| \le |1/d|$, it suffices to prove

$$\left|1/d\right| \times \left|\left(ph_0 - sd_2 \times d \times ph_0\right)\right| < 2^{e(q_0)-24}. \quad (12)$$

But, we can show

LEMMA 7.2.3. $\left|p_0 - sd_2 \times d \times ph_0\right| < 2^{e(p_0)-26}$.

LEMMA 7.2.4. *If $p_0 \ne 0$, then* $\left|1/d\right| \times 2^{e(p_0)} \le 2^{e(q_0)+2}$.

If $p_0 = 0$, then $ph_0 = 0$ and, so, (12) is trivial. Otherwise, we have the two inequalities above. Multiplying them together and simplifying gives (12) and, so, the proof of the α bound is complete. That, in turn, means that the proof of the separation property for q_0 and q_1 is complete. □

The proofs of Lemmas 7.2.3 and 7.2.4 are left for the reader. Hint: Use Lemma 7.1.1 to bound the relative error in sd_2, expand the definitions of ph_0 and q_0, and appeal to the fundamental properties of trunc and away.

8 Proof of Theorem 2

THEOREM 2. *If p and d are* 15,,64 *floating-point numbers, $d \ne 0$ and mode is a rounding mode, then* divide!(p, d, *mode*) = divide(p, d, *mode*).

8.1 The Numeric Equivalence Lemmas

The proof of Theorem 2 proceeds by showing that each numeric semantic function is equivalent to the corresponding semantic function. For example,

LEMMA 8.1.1. *If d is a* 15,,64 *floating-point number and $d \ne 0$, then* $\text{sd}_2(d) = \text{esd}_2(d)$.

We call this theorem the "numeric equivalence lemma" for sd_2. If we prove such a lemma, e.g., $v = ev$, for each of lines 1-32, (adding appropriate hypotheses for p, d, and *mode* when necessary), then the proof of Theorem 2 is easy.

PROOF. Consider divide(p, d, *mode*). By definition, it is round($\text{eqq}_1(p, d) + \text{eq}_0(p, d)$, *mode*), provided eqq_1 and all other semantic functions yield numeric values. But, by the numeric equivalence lemmas for lines 1-31, this provision is met and $\text{eqq}_1(p, d) + \text{eq}_0(p, d)$ is $\text{qq}_1(p, d) + \text{q}_0$. Hence, divide($p$, d, *mode*) = round($\text{qq}_1(p, d) + \text{qq}_0(p, d)$, *mode*) = divide!($p$, d, *mode*). □

Now, consider the proofs of the first 31 numeric equivalence lemmas. From the definition of the semantic functions and the numeric equivalence lemmas preceding that for v, we know that $v = ev$ if v is a 17,,n normalized floating-point number. So, we are merely obliged to prove that each v (other than divide!) is a floating-point number of the desired precision and each has a 17-bit exponent. We deal with precision first and, then, look at the exponent bounds.

TABLE 3
EXPONENT BOUNDS FOR LINES 7 THROUGH 32

7.					e_d	$=$	$e(dh)$	$=$	e_d
8.	dl	$\neq$	0	$\rightarrow$	$e_d - 63$	$\leq$	$e(dl)$	$\leq$	$e_d - 31$
9.					e_p	$=$	$e(p_0)$	$=$	e_p
10.					e_p	$=$	$e(ph_0)$	$=$	e_p
11.	p	$\neq$	0	$\rightarrow$	$e_p - e_d - 2$	$\leq$	$e(q_0)$	$\leq$	$e_p - e_d + 3$
12.	qdh_0	$\neq$	0	$\rightarrow$	$e_p - 2$	$\leq$	$e(qdh_0)$	$\leq$	$e_p + 4$
13.	qdl_0	$\neq$	0	$\rightarrow$	$e_p - 65$	$\leq$	$e(qdl_0)$	$\leq$	$e_p - 27$
14.	pt_1	$\neq$	0	$\rightarrow$	$e_p - 65$	$\leq$	$e(pt_1)$	$\leq$	$e_p + 5$
15.	p_1	$\neq$	0	$\rightarrow$	$e_p - 128$	$\leq$	$e(p_1)$	$\leq$	$e_p + 6$
16.	p_1	$\neq$	0	$\rightarrow$	$e_p - 128$	$\leq$	$e(ph_1)$	$\leq$	$e_p + 6$
17.	q_1	$\neq$	0	$\rightarrow$	$e_p - e_d - 130$	$\leq$	$e(q_1)$	$\leq$	$e_p - e_d + 9$
18.	qdh_1	$\neq$	0	$\rightarrow$	$e_p - 130$	$\leq$	$e(qdh_1)$	$\leq$	$e_p + 10$
19.	qdl_1	$\neq$	0	$\rightarrow$	$e_p - 193$	$\leq$	$e(qdl_1)$	$\leq$	$e_p - 21$
20.	pt_2	$\neq$	0	$\rightarrow$	$e_p - 193$	$\leq$	$e(pt_2)$	$\leq$	$e_p + 11$
21.	p_2	$\neq$	0	$\rightarrow$	$e_p - 256$	$\leq$	$e(p_2)$	$\leq$	$e_p + 12$
22.	p_2	$\neq$	0	$\rightarrow$	$e_p - 256$	$\leq$	$e(ph_2)$	$\leq$	$e_p + 12$
23.	q_2	$\neq$	0	$\rightarrow$	$e_p - e_d - 258$	$\leq$	$e(q_2)$	$\leq$	$e_p - e_d + 15$
24.	qdh_2	$\neq$	0	$\rightarrow$	$e_p - 258$	$\leq$	$e(qdh_2)$	$\leq$	$e_p + 16$
25.	qdl_2	$\neq$	0	$\rightarrow$	$e_p - 321$	$\leq$	$e(qdl_2)$	$\leq$	$e_p - 15$
26.	pt_3	$\neq$	0	$\rightarrow$	$e_p - 321$	$\leq$	$e(pt_3)$	$\leq$	$e_p + 17$
27.	p_3	$\neq$	0	$\rightarrow$	$e_p - 384$	$\leq$	$e(p_3)$	$\leq$	$e_p + 18$
28.	p_3	$\neq$	0	$\rightarrow$	$e_p - 384$	$\leq$	$e(ph_3)$	$\leq$	$e_p + 18$
29.	q_3	$\neq$	0	$\rightarrow$	$e_p - e_d - 386$	$\leq$	$e(q_3)$	$\leq$	$e_p - e_d + 21$
30.	p_2	$\neq$	0	$\rightarrow$	$e_p - e_d - 409$	$\leq$	$e(qq_2)$	$\leq$	$e_p - e_d + 22$
31.	p_1	$\neq$	0	$\rightarrow$	$e_p - e_d - 472$	$\leq$	$e(qq_1)$	$\leq$	$e_p - e_d + 23$
32.	p	$\neq$	0	$\rightarrow$	$e_p - e_d - 535$	$\leq$	$e(divide!)$	$\leq$	$e_p - e_d + 25$

8.2 Precision

The precision analysis is interesting only for those lines of code containing an exactness assertion. (Lines containing a rounding mode are trivial to handle because the corresponding numeric function is defined to round to the desired precision.) So, consider, say, line 15 of Fig. 1, where we must prove that p_1 fits in 64 bits, which is to say $\text{trunc}(p_1, 64) = p_1$. Recall here we are dealing with the numeric functions and we know $p_1 = p_0 - q_0 \times d$. To prove this and the related partial remainder theorems, we appeal to the general lemma:

LEMMA 8.2.1. *If p, d, and q are nonzero rationals such that* $\text{trunc}(p, 64) = p$, $\text{trunc}(d, 64) = d$, $\text{trunc}(q, 24) = q$, *and* $|p - q \times d| \leq |d| \times 2^{e(q)-23}$, *then* $\text{trunc}(p - q \times d, 64) = p - q \times d$.

We leave the proof to the reader. Hint: Consider whether $q \times d$ fits in 87 bits and use Lemmas 6.2.10 and 6.2.11.

8.3 Exponents

We must also show that each variable satisfies the exponent requirements on 17,,n normalized floating-point numbers, provided p and d are 15,,64 floating-point numbers and *mode* is a rounding mode. Thus, we may assume $-62 - 2^{14} \leq e_p \leq 2^{14}$ and $-62 - 2^{14} \leq e_d \leq 2^{14}$ and we must prove that the exponent of each code variable v satisfies $1 - 2^{16} \leq e(v) \leq 2^{16}$.

Our proof considers sequentially the numeric interpretation of each variable v and bounds $e(v)$ in terms of e_p and e_d.

Given the work we did for Lemma 7.1.1, the first six lines are straightforward. For example, it is easy to show $-e_d - 2 \leq e(sd_2) \leq -e_d + 1$. Hence, if $-62 - 2^{14} \leq e_d \leq 2^{14}$, then it is easy to show $1 - 2^{16} \leq e(sd_2) \leq 2^{16}$.

The remaining lines are handled by the regular application of the elementary lemmas plus

LEMMA 8.3.1. *If x and y are rationals such that $x + y \neq 0$, n and m are positive integers,* $\text{trunc}(x, n) = x$ *and* $\text{trunc}(y, m) = y$, *then* $1 - \max(n, m) + \min(e_x, e_y) \leq e(x + y)$.

LEMMA 8.3.2. *If x and y are nonzero rationals whose sum is nonzero, then* $e(x + y) \leq 1 + \max(e_x, e_y)$.

While these lemmas provide rather sloppy bounds, we can tolerate the sloppiness because exponents of width 15 (even taking into account the small expansion due to denormalization) are so much smaller than those of width 17. We can deduce the theorems in Table 3 from which our goals follow.

9 CONCLUDING REMARKS

The success of this work, and the work cited in [21], [22], [31], [35], [23] indicates that today's general-purpose mechanical theorem provers can be used to check the proofs of complex commercial algorithms. However, the method requires additional input from an expert who understands both the application and the underlying mechanized reasoning tool. Much of this additional input consists of details which would otherwise be considered obvious, and sometimes the proof of the obvious is difficult. In this paper, we have outlined how such experts have guided ACL2 to the theorem about the AMD5$_K$86 division program.

More specifically, provided that the prover was sound and used correctly, we have proven that if p and d are 15,,64 floating-point numbers, $d \neq 0$, and *mode* is a rounding mode, *divide* returns the result of rounding the "infinitely precise" quotient p/d with the user-specified *mode*. Given this, if the input *proof script* given to the prover accurately describes

the functionality of the hardware, then the theorem about *divide* also applies to the AMD5$_K$86 division program. Furthermore, if the descriptions in the proof script have correctly captured the salient aspects of the IEEE std. 754, the AMD5$_K$86 division program is compliant.

Few would disagree that an implementation of an algorithm without any proofs is precarious. Bugs can occur anywhere between the point of conception and the point the design is cast. Some of these points are completely cognitive; others are only described in paper documents. Even if a *golden model* exists, using it to test a black box probably will not instill the level of confidence needed to justify inclusion of a design in a modern microprocessor. Also, such approaches give little insight into how to partition the problem. However, after a proof is discovered, the situation becomes more focused. A bug can only exist if there is a difference between the functionality of the hardware and the functionality described by the proof script, an important aspect or property was left out of the theorem, there was a mistake in the proof, or, quite simply, the functional goal was not actually what was needed. These points can then be attacked separately.

It was our experience that the introduction of a theorem prover to this status quo of having a hand proof focused the verification problem further. As soon as ACL2 certified that the proof script described a theorem, we had traded the probability of a bug being in the hand proof for the probability that the prover itself had a bug that harmed the result. Furthermore, we gained an executable record of the assumptions made about unit functionality, and a mechanism by which the proof could be modified and instantly "replayed."[2] Replaying the proof is necessary when changes are made as a result of attacking the other verification points.

Indeed, our original proof script was changed several times before this presentation, and each time a change was made, Moore replayed the script to see if *divide* was still p/d. Changes have been slight (e.g., narrowing by 1 the bounds of a representable exponent) and, so far, none have exposed a bug. The source of every change thus far has been at the point of discrepancy between the functionality of the hardware units and their functional descriptions in the proof script—and we believe that forcing this correspondence is technically feasible.

Thus, having a mechanically verified numerical proof of the divide microcode raised the level of confidence we had gained from conventional methods. But, perhaps more importantly, it is now clear that the development and application of more tools will further increase our confidence in future implementations.

Acknowledgments

This work was supported by Advanced Micro Devices and performed in conjunction with Computational Logic Incorporated. It was done with the assistance of Ashraf Ahmed, Mike Schulte, Tom Callaway, Kelvin Koveas, Mike Goddard, and others. The project was assembled with the help of Warren Hunt, Terry Hulett, Dave Reed, and Chuck Colburn. The final form of this paper is due in large part to input from the anonymous referees. A the time of this work, J Strother Moore and Matt Kaufmann were employed at Computational Logic, Inc., Austin, Texas, and Tom Lynch was employed at Advanced Micro Devices, Austin, Texas. The theorem prover used in this work was supported in part at Computation Logic, Inc., by the U.S. Defense Advanced Research Projects Agency, ARPA Order 7406, and the U.S. Office of Naval Research, Contract N00014-94-C-0193. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Advanced Micro Devices, Inc., Computational Logic, Inc., the U.S. Defense Advanced Research Projects Agency, the U.S. Office of Naval Research, or the U.S. government.

References

[1] D. Shephard, "Using Mathematical Logic and Formal Methods to Write Correct Microcode," *SIGMICRO Newsletter*, vol. 19, nos. 1-2, pp. 60-64, June 1988.

[2] M.K. Srivas and S.P. Miller, "Applying Formal Verification to a Commercial Microprocessor," *Proc. ASP-DAC95/CHDL95/VLSI95 Asia and South Pacific Design Automation Conf.*, pp. 493-502, Aug. 1995.

[3] A. Camilleri, M. Gordon, and T. Melham, "Hardware Verification Using Higher Order Logic," *HDL Descriptions to Guaranteed Correct Circuit Designs*. Elsevier, 1987.

[4] W.A. Hunt. "Microprocessor Design Verification." *J. Automated Reasoning*, vol. 5, no. 4, pp. 429-460, 1989.

[5] T. Lynch, A. Ahmed, M. Schulte, T. Callaway, and R. Tisdale. "The K5 Transcendental Functions," *Proc. 12th Symp. Computer Arithmetic*, pp. 163-170, Bath, England, 1995.

[6] S. Coupet-Grimal and L. Jakubiec, "Coq and Hardware Verification: A Case Study," *Theorem Proving and Higher Order Logics*, pp. 125-139. Springer-Verlag, 1996.

[7] D.E. Atkins, "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders," *IEEE Trans. Computers*, vol. 17, p. 930, Oct., 1968.

[8] C.S. Wallace, "Suggested Design for a Fast Multiplier," *IEEE Trans. Electronic Computers*, vol. 13, pp. 14-17, 1964.

[9] D. Ferrari, "A Division Method Using a Parallel Multiplier," *IEEE Trans. Electronic Computers*, vol. 16, pp. 224-226, Apr. 1967.

[10] M.J. Schulte, "Optimal Approximations for the Newton-Raphson Division Algorithm," *Proc. SCAN-93 Conf. Scientific Computing, Computer Arithmetic, and Numeric Validation*, Vienna, Sept. 1993.

[11] W.B. Brigs and D.W. Matula, "Method and Apparatus for Performing Division Using a Rectangular Aspect Ratio Multiplier," U.S. Patent No. 5046038, 1991.

[12] D. DasSarma and D.W. Matuala, "Measuring the Acuracy of ROM Reciprocal Tables," *IEEE Trans. Computers*, vol. 42, no. 8, pp. 932-940, Aug. 1994.

[13] W.F. Wong and E. Goto, "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multiplies," *IEEE Trans. Computers*, pp. 278-294, Mar. 1994.

[14] D. Wong and M. Flynn, "Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations," *IEEE Trans. Computers*, pp. 981-995, Aug. 1992.

[15] D.M. Priest, "Algorithms for Arbitrary Precision Floating Point Arithmetic," *Proc. 10th Symp. Computer Arithmetic*, pp. 132-153, Grenoble, France, 1991.

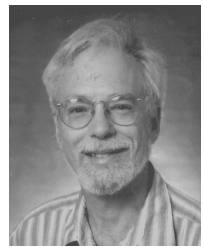
[16] J.H. Wilkinson, *Rounding Errors in Algebraic Process*. London: Her Majesty's Stationary Office, 1963.

[17] P.H. Sterbenz, *Floating-Point Computation*. Englewood Cliffs, N.J.: Prentice Hall, 1974.

[18] J.B. Wilson, "An Algorithm for Rapid Binary Division," *IEEE Trans. Electronic Computers*, vol. 10, pp. 662-670, Dec. 1961.

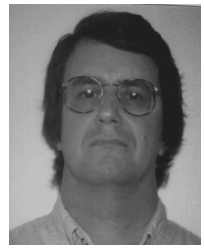
2. Replaying the *divide* proof script takes approximately two hours on a Sun Microsystems Ultra 2-2170.

[19] E. Ukkonen, "On the Calculation of the Effects of Roundoff Errors," *ACM Trans. Mathematical Software*, vol. 7, no. 3, pp. 259-271, Sept. 1981.
[20] R.S. Boyer, M. Kaufmann, and J S. Moore, "The Boyer-Moore Theorem Prover and Its Interactive Enhancement," *Computers and Mathematics with Applications*, vol. 29, no. 2, pp. 27–62, 1995.
[21] R.S. Boyer and J S. Moore, *A Computational Logic Handbook*. New York: Academic Press, 1988.
[22] R.S. Boyer and Y. Yu, "Automated Proofs of Object Code for a Widely Used Microprocessor," *J. ACM*, vol. 43, no. 1, pp. 166-192, Jan. 1996.
[23] B. Brock, M. Kaufmann, and J S. Moore, "ACL2 Theorems about Commercial Microprocessors," *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas and A. Camilleri, eds. , pp. 275–293. Springer-Verlag, Nov. 1996.
[24] R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," *ACM Computing Surveys*, vol. 40, no. 3, pp. 293-318, Sept. 1992.
[25] R.E. Bryant, "Bit-Level Analysis of an SRT Divider Circuit," Technical Report CMU-CS-95-140, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Penn.
[26] D. Christie, "Developing the AMD-K5 Architecture," *IEEE Micro*, pp. 16-26, April, 1996.
[27] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A Tutorial Introduction to PVS," presented at *Workshop Industrial-Strength Formal Specification Techniques*, Boca Raton, Fla., Apr. 1995 (see http://www.csl.sri.com/pvs.html).
[28] M.D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Norwell, Mass.: Kluwer Academic, 1994.
[29] S.F. Oberman and M.J. Flynn, "Design Issues in Division and Other Floating-Point Operations," *IEEE Trans. Computers*, vol. 46, no. 2, pp. 154-161, Feb. 1997.
[30] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5-48, Mar. 1991.
[31] W.A. Hunt Jr. and B. Brock, "A Formal HDL and Its Use in the FM9001 Verification," *Proc. Royal Soc.*, 1992.
[32] M. Kaufmann and J S. Moore, *ACL2 Version 2.1*, http://www.cs.utexas.edu/ users/moore/acl2, 1998.
[33] M. Kaufmann and J S. Moore, "ACL2: An Industrial Strength Theorem Prover for a Logic Based on Common LISP," *IEEE Trans. Software Eng.*, vol. 23, no. 4, pp. 203-213, Apr. 1997.
[34] T. Lynch, A. Ahmed, and M. Schulte, "Rounding Error Analysis for Division," technical report, Advanced Micro Devices, Inc., 5204 East Ben White Blvd., Austin, TX 78741, 26 May 1995.
[35] S.P. Miller and M. Srivas, "Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods," *Proc. WIFT '95: Workshop Industrial-Strength Formal Specification Techniques*, pp. 2-16, Apr. 1995.
[36] P.M. Miner, "Defining the IEEE-854 Floating-Point Standard in PVS," NASA Technical Memorandum 110167, NASA Langley Research Center, Hampton, Va., 1995.
[37] P.M. Miner, "Verification of IEEE Compliant Subtractive Division Algorithms," *Formal Methods in Computer-Aided Design (FMCAD'96)*, M. Srivas and A. Camilleri, eds., pp. 64-78. Springer-Verlag, Nov. 1996.
[38] D.A. Patterson and J.L. Hennessy, *Computer Architecture*. San Mateo, Calif.: Morgan Kaufmann Publishers, 1990.
[39] Standards Committee of the IEEE Computer Society, *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std. 754-1985, 1985.
[40] Standards Committee of the IEEE Computer Society. *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, IEEE Std. 854-1987, 1987.
[41] H. Rueß, M.K. Srivas, and N. Shankar, "Modular Verification of SRT Division," Computer Science Laboratory, SRI Int'l, Menlo Park, Calif., 1996.
[42] D. Russinoff, "A Mechanically Checked Proof of Correctness of the AMD5$_K$86 Floating-Point Square Root Microcode," http://www.onr.com/user/russ/david/fsqrt.html, Feb. 1997.
[43] D. Rusinoff, "A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD K7 Floating-Point Division and Square Root Instructions," http://www.onr.com/user/russ/david/k7-div-sqrt, Feb. 1998.
[44] G.L. Steele Jr., *Common LISP: The Language*. Bedford, Mass.: Digital Press, 1984.
[45] G.L. Steele Jr., *Common Lisp The Language, second ed*. Burlington, Mass.: Digital Press, 1990.



J Strother Moore received his PhD from the University of Edinburgh in 1973 and his BS from the Massachusetts Institute of Technology in 1970. Dr. Moore holds the Admiral B.R. Inman Centennial Chair in Computing Theory at the University of Texas at Austin. He is the author of many books and papers on automated theorem proving and mechanical verification of computing systems, including *Piton: A Mechanically Verified Assembly-Level Language* (Kluwer, 1996) and *A Computational Logic Handbook, Second Edition* with Robert S. Boyer (Academic Press, 1998). Along with Boyer, he is a coauthor of the Boyer-Moore theorem prover and the Boyer-Moore fast string searching algorithm. With Matt Kaufmann, he is the coauthor of the ACL2 theorem prover. He and Robert Boyer were awarded the 1991 Current Prize in Automatic Theorem Proving by the American Mathematical Society. He is a fellow of the American Association for Artificial Intelligence.

Thomas W. Lynch received a BSEE in computer engineering from the University of Texas at Austin in 1986, and an MSEE in computer engineering in 1996. Lynch was a design engineer, a member of the technical staff, and a consultant for AMD between 1986 and 1996. During that period, He was involved in various capacities in the development of eight different microprocessors for signal processing and general computing. He has authored numerous patents. In 1994, he led the development of the numeric microcode for the AMD5$_K$86 microprocessor, and established a floating-point unit verification strategy which included the use of proofs (manual and automated) to augment conventional testing. He is currently the chief technical officer of a startup that is developing tools for the functional verification of hardware and software.



Matt Kaufmann was trained as a mathematical logician at the University of Wisconsin where he received his PhD in mathematics in 1978, at which time he joined the mathematics faculty at Purdue University. After authoring numerous papers in mathematical logic, he joined Burroughs Corp. in Austin, Texas, in 1984, focusing on functional programming. From 1986 until 1995, he worked at the University of Texas and Computational Logic, Inc., on general-purpose automated reasoning, in particular co-developing the ACL2 theorem prover. Dr. Kaufmann worked on formal hardware verification and related activities at Motorola for the next two years, in particular supporting and enhancing their model-checker. He has been working on the Year 2000 problem at EDS since August 1997, where, among other things, he continues to apply formal verification techniques, this time using the ACL2 theorem prover to debug Cobol transformations and formally verify their correctness.