A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5$_K$86 ^{TM} Floating-Point Division Algorithm

J Strother Moore, Tom Lynch and Matt Kaufmann

March, 1996

Abstract

We describe a mechanically checked proof of the correctness of the kernel of the floating point division algorithm used on the AMD5$_K$86 microprocessor. The kernel is a non-restoring division algorithm that computes the floating point quotient of two double extended precision floating point numbers, p and d ($d \neq 0$), with respect to a rounding mode, *mode*. The algorithm is defined in terms of floating point addition and multiplication. First, two Newton-Raphson iterations are used to compute a floating point approximation of the reciprocal of d. The result is used to compute four floating point quotient digits in the 24,,17 format (24 bits of precision and 17 bit exponents) which are then summed using appropriate rounding modes. We prove that if p and d are 64,,15 (possibly denormal) floating point numbers, $d \neq 0$ and *mode* specifies one of six rounding procedures and a desired precision $0 < n \leq 64$, then the output of the algorithm is p/d rounded according to *mode*. We prove that every intermediate result is a floating point number in the format required by the resources allocated to it. Our claims have been mechanically checked using the ACL2 theorem prover.

1 The Algorithm

The floating point division algorithm, called "divide" and shown in Figure 1, takes three inputs: floating point numbers p and d and a "rounding mode" *mode*. The algorithm uses

Algorithm divide($p, d, mode$)

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

Figure 1: The Division Algorithm

a lookup table which we discuss after we have explained how to read the "pseudocode" in which the algorithm is expressed.

The algorithm is proved correct only when p and d are 64,,15 (possibly denormal) floating point numbers. We define the $n, , m$ notation on page 11; informally, n is the number of bits of precision and m is the maximum number of bits in the exponent.

1.1 The Pseudocode Semantics

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

$i.\ \ var = expr$ *pair*   ;*comment*

where i is a line number, *var* is a variable symbol, *expr* is a mathematical expression, *pair* is either a "rounding mode" of the form `[`*name* *n*`]` or an "exactness claim" of the form `[exact` *n*`]`, and *comment* is a comment. The expression involves previously mentioned variable symbols and the familiar operations of addition ($+$), subtraction ($-$), and multiplication ($\times$) of rational numbers, table lookup (in the assignment to sd_0 on line 1) and ones complement (denoted by comp in the assignments to sd_1 and sd_2 on lines 4 and 6). If the variables involved in an expression have rational values, then the expression has a well-defined mathematical value and that value is rational.

The execution of such a statement assigns to the variable *var* the result eround($val, pair$), where *val* is the value of *expr* (under the preceding assignments). The function eround is defined on page 14 and can be informally described as follows. If *pair* is a rounding mode, eround($val, pair$) rounds *val* to n bits of precision according to *pair* and checks that the exponent of the rounded result fits in 17 bits. If so, eround returns the rounded result; otherwise, eround returns an "error object." If *pair* is an exactness claim, eround checks that *val* has at most n bits of precision and a 17 bit exponent and returns *val* or an error object accordingly.

The value computed by the algorithm is determined by executing each of the assignment statements, sequentially, and returning either the first error object assigned to a variable, if any, or the value of the variable *divide*.

1.2 The Lookup Table

The function lookup maps a 64,,17 floating point number d into an approximation of $1/d$. The approximation is an 8,,17 floating point number.

$$\mathrm{lookup}(d) \quad = \quad \sigma_d \times \mathrm{table}(\mathrm{s}_d) \times 2^{-\mathrm{e}_d},$$

where σ_d, s_d and e_d are the sign, significand, and exponent, respectively, of d (see Section 5). By $\mathrm{table}(\mathrm{s}_d)$ above we mean the entry associated with the most significant 8 bits of s_d (i.e., $\mathrm{truncn}(\mathrm{s}_d, 8)$, page 11) in Table 1. The table maps each of the 128 8-bit non-0 significands to an 8-bit approximation of its reciprocal. The computation of the table entries is discussed in [8].

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
1.0000011_2	0.11111001_2	1.0100011_2	0.11001000_2	1.1000011_2	0.10101000_2	1.1100011_2	0.10010000_2
1.0000100_2	0.11110111_2	1.0100100_2	0.11000111_2	1.1000100_2	0.10100111_2	1.1100100_2	0.10001111_2
1.0000101_2	0.11110101_2	1.0100101_2	0.11000110_2	1.1000101_2	0.10100110_2	1.1100101_2	0.10001111_2
1.0000110_2	0.11110100_2	1.0100110_2	0.11000101_2	1.1000110_2	0.10100101_2	1.1100110_2	0.10001110_2
1.0000111_2	0.11110010_2	1.0100111_2	0.11000100_2	1.1000111_2	0.10100100_2	1.1100111_2	0.10001110_2
1.0001000_2	0.11110000_2	1.0101000_2	0.11000010_2	1.1001000_2	0.10100011_2	1.1101000_2	0.10001101_2
1.0001001_2	0.11101110_2	1.0101001_2	0.11000001_2	1.1001001_2	0.10100011_2	1.1101001_2	0.10001100_2
1.0001010_2	0.11101101_2	1.0101010_2	0.11000000_2	1.1001010_2	0.10100010_2	1.1101010_2	0.10001100_2
1.0001011_2	0.11101011_2	1.0101011_2	0.10111111_2	1.1001011_2	0.10100001_2	1.1101011_2	0.10001011_2
1.0001100_2	0.11101001_2	1.0101100_2	0.10111110_2	1.1001100_2	0.10100000_2	1.1101100_2	0.10001011_2
1.0001101_2	0.11101000_2	1.0101101_2	0.10111101_2	1.1001101_2	0.10011111_2	1.1101101_2	0.10001010_2
1.0001110_2	0.11100110_2	1.0101110_2	0.10111100_2	1.1001110_2	0.10011111_2	1.1101110_2	0.10001001_2
1.0001111_2	0.11100100_2	1.0101111_2	0.10111011_2	1.1001111_2	0.10011110_2	1.1101111_2	0.10001001_2
1.0010000_2	0.11100011_2	1.0110000_2	0.10111010_2	1.1010000_2	0.10011101_2	1.1110000_2	0.10001000_2
1.0010001_2	0.11100001_2	1.0110001_2	0.10111001_2	1.1010001_2	0.10011100_2	1.1110001_2	0.10001000_2
1.0010010_2	0.11100000_2	1.0110010_2	0.10111000_2	1.1010010_2	0.10011100_2	1.1110010_2	0.10000111_2
1.0010011_2	0.11011110_2	1.0110011_2	0.10110111_2	1.1010011_2	0.10011011_2	1.1110011_2	0.10000111_2
1.0010100_2	0.11011101_2	1.0110100_2	0.10110110_2	1.1010100_2	0.10011010_2	1.1110100_2	0.10000110_2
1.0010101_2	0.11011011_2	1.0110101_2	0.10110101_2	1.1010101_2	0.10011001_2	1.1110101_2	0.10000101_2
1.0010110_2	0.11011010_2	1.0110110_2	0.10110100_2	1.1010110_2	0.10011001_2	1.1110110_2	0.10000101_2
1.0010111_2	0.11011000_2	1.0110111_2	0.10110011_2	1.1010111_2	0.10011000_2	1.1110111_2	0.10000100_2
1.0011000_2	0.11010111_2	1.0111000_2	0.10110010_2	1.1011000_2	0.10010111_2	1.1111000_2	0.10000100_2
1.0011001_2	0.11010101_2	1.0111001_2	0.10110001_2	1.1011001_2	0.10010111_2	1.1111001_2	0.10000011_2
1.0011010_2	0.11010100_2	1.0111010_2	0.10110000_2	1.1011010_2	0.10010110_2	1.1111010_2	0.10000011_2
1.0011011_2	0.11010011_2	1.0111011_2	0.10101111_2	1.1011011_2	0.10010101_2	1.1111011_2	0.10000010_2
1.0011100_2	0.11010001_2	1.0111100_2	0.10101110_2	1.1011100_2	0.10010101_2	1.1111100_2	0.10000010_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

Table 1: Inverse Table

2 The Theorem

We have mechanically checked a proof of the following theorem, stated informally below and formally in Section 5.

Theorem: *If p and d are 64,,15 (possibly denormal) floating point numbers, $d \neq 0$, and mode is a rounding mode of the form* `[`*name n*`]`*, where name* $\in$ {`trunc`, `away`, `sticky`, `nearest`, `posinf`, `neginf`} *and n is an integer, $0 < n \leq 64$, then the algorithm yields the n,,17 floating point number obtained by dividing p by d and rounding the precise mathematical result according to mode.*

A corollary is that when p, d, and *mode* satisfy the conditions of the theorem, the algorithm never signals an error. Hence, all the intermediate results fit in floating point registers of the size indicated by the *pair* field of each assignment.

Suppose one has hardware that implements floating point addition and multiplication with respect to a rounding mode as done in the IEEE floating point standard. The algorithm discussed here shows an implementation of the division operation with respect to a rounding mode. Our theorem establishes that the algorithm returns the result of rounding the infinitely precise answer according to the specified mode.

We do not here deal with the NaNs, infinities and signed zeros of the IEEE standard; we discuss how denormalized numbers are handled later.

3 How the Algorithm Works

3.1 Some Illustrative Examples in Decimal Notation

Consider the familiar long division algorithm. In Figure 2 we show the long division calculation of 430 divided by 12.

Very loosely, the long division of p by d can be thought of as generating a sequence of n quotient digits, q_i and partial remainders, p_i, where $p = p_0$, such that the final quotient is the sum of the n quotient digits and p_n is the final remainder.

- Given p_i and d, guess the next quotient digit, q_i, by inspection of the high order parts of p_i and d. In dividing 430 by 12 in Figure 2 our q_0 is the digit 3 in the tens place, or 30. In long division one generally tries to choose q_i to make the next partial remainder nonnegative and minimal but there is quite a lot of flexibility in the choice of q_i.

- To obtain the next partial remainder, p_{i+1}, multiply q_i times d and subtract the result from p_i. In the example of 430 divided by 12 with a q_0 of 30, the first partial remainder is $430 - (12 \times 30) = 70$.

Let q be the sum of the n quotient digits. Let r be the last partial remainder, p_n. Then $p = dq + r$. If one chooses appropriate q_i and n, $|r|$ can be made to approach 0.

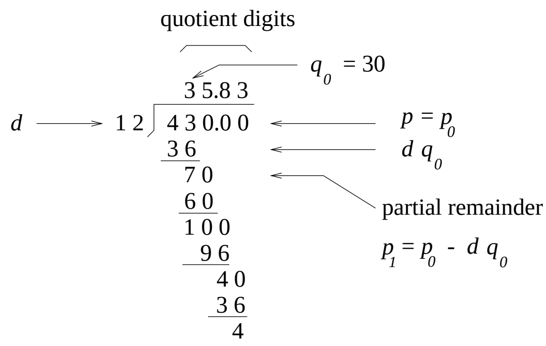

Figure 2: The Nomenclature of Long Division

In Figure 3 we show the same long division problem, except this time we generalize the quotient digits to two (decimal) digit floating point numbers and we show how the reciprocal of the divisor is used to generate the quotient digits using fixed precision operations.

The reciprocal of 12 is $0.08\overline{3}$ which we approximate to two decimal digits of precision with 0.083. We call this quantity sd_2 because its use in this decimal example is analogous to the variable sd_2 in the binary floating point algorithm of Figure 1.

The basic idea behind the computation of the quotient digits is to let q_i be the high order part of $p_i/d = (1/d)p_i$. The first quotient digit, q_0, is obtained by multiplying sd_2 by 430, obtaining 35.69, which is rounded to two digits. In this example `away` rounding is used. (In

```
        36.
   +     -.17
   +      .0034
   +     -.000067
        ---------
        35.833333
12 ) 430.000000
     432.
       -2.
       -2.04
         .04
         .0408
       - .0008
       - .000804
         .000004
```

Reciprocal Calculation:
$1/12 \;=\; 0.083\overline{3} \approx 0.083 = sd_2$

Quotient Digit Calculation:

$$\begin{array}{lrclcrcl}
0.083 \times & 430.0000 & = & 35.6900000 & \approx & 36.000000 & = & q_0 \\
0.083 \times & -2.0000 & = & -.1660000 & \approx & -.170000 & = & q_1 \\
0.083 \times & .0400 & = & .0033200 & \approx & .003400 & = & q_2 \\
0.083 \times & -.0008 & = & -.0000664 & \approx & -.000067 & = & q_3
\end{array}$$

Summation of Quotient Digits:
$q_0 + q_1 + q_2 + q_3 = 35.833333$

Figure 3: Quotient Digits Can be Negative

the actual algorithm, `away` rounding is used on all but the last digit, where `trunc` rounding is used.) Thus, q_0 here is 36. This is slightly too big: the next partial remainder, p_1, is -2.0, so the next quotient digit, q_1, is negative (-.17 after `away` rounding to two digits) and so tends to correct the over-estimation of the first digit. The sum, q, of the four quotient digits is 35.833333. Observe that a correct answer is still obtained: 430 divided by 12 produces 35.833333 with a remainder of 0.000004.

3.2 The Reciprocal Computation

Now consider the algorithm in Figure 1. Lines 1 through 6 are devoted to the computation of the reciprocal of d. At line 6 the variable sd_2 is assigned a 32,,17 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 via Table 1 and then refining it 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 mode `[away 32]`, i.e., we use a 32,,17 floating point number approximating the 64,,17 number d from above.

- After multiplying sd_i by (the approximation to) d with a 32×32-bit floating point multiply, 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 $(2 - sdd_i - 2^{-31})$ or $(2 - sdd_i - 2^{-32})$, depending on whether $sdd_i < 1$. This notion of "complement" is explained in Subsection 5.1.
- After multiplying by sd_i we truncate the result to 32 bits.

3.3 The Quotient Digit Computation

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 24,,17 floating point number. Quotient digit q_i is obtained by multiplying sd_2 by the top 32-bits of p_i (a 32×32-bit operation) and rounding to 24 bits. See lines 11, 17, 23, and 29. In the computations of the first three quotient digits we use `away` rounding, but in the last quotient digit we use `trunc`.

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 in two 32-bit pieces. In particular, dh and dl are defined (lines 7 and 8) to be the high and low 32 bits of d. Thus $(q_i \times d) = q_i \times dh + q_i \times dl$. Each product, called qdh_i and qdl_i, can be computed exactly with a 24×32-bit multiplication. See for example lines 12 and 13. We then subtract these two products successively from p_i to obtain p_{i+1} in a 64×64-bit operation. The 64,,17 floating point result is exact; no precision is lost and p_{i+1} as computed is indeed the mathematical $p_i - (q_i \times d)$.

3.4 Summing the Quotient Digits

In lines 30 through 32 the quotient digits are summed. The two least significant digits, q_2 and q_3 are added together first. The result is `sticky` rounded to 64 bits and called qq_2. Then q_1 is added to qq_2, `sticky` rounded to 64 bits and called qq_1. Finally, q_0 is added to qq_1, but this time the result is rounded according to the user-specified *mode* and returned as the quotient.

4 The ACL2 Logic and Theorem Prover

Our mechanical proof of the correctness of this algorithm was carried out with ACL2. "ACL2" stands for "A Computational Logic for Applicative Common Lisp." It provides a formal mathematical logic and a mechanical theorem prover to help the user construct proofs of theorems in the logic. Because our work was done formally within that logic, the logic has influenced the style of our formalization. To help the reader understand that style we introduce the ACL2 system briefly here. However, this section can be skipped: the mathematics as presented in this paper is accessible to those with no knowledge of ACL2.

The logic formalizes a useful applicative subset of the ANSI standard programming language Common Lisp [14, 15]. Expressions in the logic are Common Lisp expressions. Given values for the variables, logical expressions can evaluated by Common Lisp implementations. Technically, the ACL2 logic is a first-order, quantifier-free logic of recursive functions providing mathematical induction on the ordinals up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$, recursive definition, and witnessed constraint of new function symbols. The following disjoint data types are axiomatized:

- **Symbols**. Symbols are Common Lisp objects denoting words. We display symbol constants in typewriter font. For example, `trunc` and `away` are both symbol constants.

- **Rationals and Integers**. The rational numbers are axiomatized so that the integers are a subset of the rationals. Thus, $22/7$ and -5 are both rationals; the latter is an integer. The familiar arithmetic operators are defined in accordance with the laws of rational arithmetic: e.g., $22/7 \times 1/11 = 2/7$. The logic implements "infinitely precise" rational arithmetic.

- **Lists**. ACL2 supports arbitrary ordered pairs and lists of ACL2 objects. For example, `[away 24]` is a list of length two containing the symbol `away` and the integer `24`.

- **Characters and Strings**. ACL2 supports many of Common Lisp's character objects and strings of characters, e.g., `"bad post-round exponent"`.

As noted, the syntax of ACL2 is that of Common Lisp. Thus, for example, the formal expression of $x \times 2^{i+1}$ in ACL2 is `(* x (expt 2 (+ i 1)))`. In this paper we hand translate all of the arithmetic expressions into traditional notation. We also use traditional mathematical English rather than formal logical notation.

ACL2 is described in more detail in [6, 7]. ACL2 is available without fee on the Internet. See `http://www.cli.com`.

Because ACL2 is an executable logic and divide is defined as a function in ACL2, it is possible to execute divide on concrete data to test it. For example, divide(1, 3, [`trunc 24`]) produces the rational 5592405/16777216 or $0.010101010101010101010101010_2$. On a Sparc Station 20 such a test of the formal pseudocode semantics takes about 0.3 seconds. If we use `away` rounding instead, the answer is $0.010101010101010101010101011_2$. In both cases the answer returned is the true quotient, 1/3, rounded as specified.

5 Formalization of the Theorem

5.1 Basic Concepts

We now explain our formalization of the floating point numbers. Every non-0 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 $1 \leq s < 2$, 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 $\sigma(x)$, $\mathrm{s}(x)$, and $\mathrm{e}(x)$. 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 $1 \leq s < 2$.

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, $1/3 = 0.01\overline{01}_2$ has a significand of $4/3 = 1.01\overline{01}_2$ and an exponent of -2, since $1/3 = 1 \times 4/3 \times 2^{-2}$ and $1 \leq 4/3 < 2$.

To truncate a significand to n bits we use:

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

For example,

$$\begin{aligned}
\mathrm{truncn}(4/3, 5) &= \mathrm{truncn}(1.0101\overline{01}_2, 5) \\
&= \lfloor 1.0101\overline{01}_2 \times 2^4 \rfloor / 2^4 \\
&= \lfloor 10101.\overline{01}_2 \rfloor / 2^4 \\
&= 10101_2 / 2^4 \\
&= 1.0101_2 \\
&= 21/16
\end{aligned}$$

We say x is an $n,,m$ *floating point number* if and only if x is a rational number, $\mathrm{truncn}(\mathrm{s}_x, n) = \mathrm{s}_x$, and $1 - 2^{m-1} \leq \mathrm{e}_x \leq 2^{m-1}$. It is helpful to think of x as "normalized" in this context.

We say x is an $n,,m^+$ *floating point number* if and only if x is a rational number, $\mathrm{truncn}(\mathrm{s}_x, n) = \mathrm{s}_x$, and $2 - n - 2^{m-1} \leq \mathrm{e}_x \leq 2^{m-1}$. In our informal discussions above when we say something is a "64,,15 (possibly denormal) floating point number" we mean it is a $64,,15^+$ floating point number.

For example, the exponent values for the 4,,3 floating point numbers are {-3, -2, ..., 3, 4}. Thus, $1.000_2 \times 2^{-3}$ is a 4,,3 floating point number. But $0.001_2 \times 2^{-3} = 1 \times 2^{-6}$ is not a 4,,3 floating point number because its exponent is -6. However the exponent range of the

$4,,3^+$ floating point numbers extends downward to -6, so $0.001_2 \times 2^{-3}$ is a $4,,3^+$ floating point number.

The reader familiar with the IEEE standard will note several discrepancies between the standard and our formal definitions above. We are not trying to formalize the standard here; we are defining a subset of the rationals on which our algorithm works.

Our notion of the $n,,m$ floating point numbers does not include the denormals, NaNs, infinities and signed zeroes of the standard. Furthermore, we allow an exponent range of $1 - 2^{m-1} \leq \mathrm{e}_x \leq 2^{m-1}$ while the standard narrows the legal exponent values to $2 - 2^{m-1} \leq \mathrm{e}_x \leq 2^{m-1} - 1$, disallowing the top and bottom values in our range. In the standard those values are appropriated for use as marks for the special cases of denormals, etc. Thus, for a given choice of n and m, our $n,,m$ floating point numbers include all of the normalized IEEE floating point numbers in that format as well as the additional rationals with exponents larger or smaller by one.

Our notion of the $n,,m^+$ floating point numbers includes all of the standard's normalized and denormal numbers in the corresponding format, as well as additional rationals. The excess in this case comes not only from our slightly larger exponent bound but also from the fact that we permit a denormal to have a full n bits of precision while the standard's denormals lose a bit of precision for each additional exponent value beyond the normal range. For example, we admit $0.001111_2 \times 2^{-3} = 1.111 \times 2^{-6}$ as a $4,,3^+$ floating point number.

We define the *complement* of an n bit floating point number x to be

$$\mathrm{comp}(x, n) = \begin{cases} 2 - x - 2^{1-n} & \text{if } x < 1 \\ 2 - x - 2^{-n} & \text{otherwise} \end{cases}$$

For example, $\mathrm{comp}(1.1110000_2, 8) = 0.00011111_2$ and $\mathrm{comp}(0.1110000_2, 8) = 1.0001111_2$. A more general notion of complement could be defined (taking into account the sign and exponent of x) but we use this one because we are only interested in the case where $1/2 < x < 2$. Indeed, we prove that both sdd_0 and sdd_1 — the arguments to comp in lines 4 and 6 of the algorithm — have this property.

We define an "ulp" ("unit in the last place") of an $n,,m$ floating point number, x, as $\mathrm{ulp}(x,n) = 2^{\mathrm{e}_x - n + 1}$. Thus, for example, $\mathrm{ulp}(1111.0001_2, 8) = 2^{3-8+1} = 0.0001_2$.

A final basic concept is that of the n^{th} bit of a significand. We index the bits by the power of 2 indicated by their positions. Because we are here thinking of s as a significand we know $0 \leq s < 2$ 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 $1 - n$.

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

For example, bit number -4 in the rational 4/3 is equal to 1. That is, $\mathrm{bitn}(4/3, -4)$ $= \mathrm{bitn}(1.010\mathbf{1}\overline{01}_2, -4) = \mathbf{1}$ because the floor expression above produces an odd integer, $\lfloor 1.010\mathbf{1}\overline{01}_2 \times 2^4 \rfloor = \lfloor 1010\mathbf{1}.\overline{01}_2 \rfloor = 1010\mathbf{1}_2$.

5.2 Rounding Procedures

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

$$\begin{aligned} \mathrm{trunc}(x,n) &= \sigma_x \times \mathrm{truncn}(\mathrm{s}_x, n) \times 2^{\mathrm{e}_x} \\ \mathrm{away}(x,n) &= \sigma_x \times \mathrm{awayn}(\mathrm{s}_x, n) \times 2^{\mathrm{e}_x} \\ \mathrm{sticky}(x,n) &= \sigma_x \times \mathrm{stickyn}(\mathrm{s}_x, n) \times 2^{\mathrm{e}_x} \end{aligned}$$

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.

$$\begin{aligned} \mathrm{awayn}(s,n) &= \begin{cases} s & \text{if } \mathrm{truncn}(s,n) = s \\ \mathrm{truncn}(s,n) + 2^{1-n} & \text{otherwise} \end{cases} \\ \mathrm{stickyn}(s,n) &= \begin{cases} s & \text{if } \mathrm{truncn}(s,n) = s \text{ or } \mathrm{bitn}(s, 1-n) = 1 \\ \mathrm{awayn}(s,n) & \text{otherwise} \end{cases} \end{aligned}$$

In addition to the rounding procedures above, we prove that the division algorithm supports three others: round to nearest, to positive infinity and to negative infinity, denoted by $\text{nearest}(x, n)$, $\text{posinf}(x, n)$ and $\text{neginf}(x, n)$. We omit their definitions here for brevity.

5.3 Rounding Modes

A *rounding mode* is a list of length two, `[`*name* *n*`]` where *name* $\in$ {`trunc`, `away`, `sticky`, `nearest`, `posinf`, `neginf`} and n is an integer such that $0 < n \leq 64$.

The general-purpose rounding operation rounds a rational according to a specified rounding mode `[`*name* *n*`]`:

$$\text{round}(x, \texttt{[}\mathit{name}\ \ n\texttt{]}) = \begin{cases} \text{away}(x, n) & \text{if } \mathit{name} = \texttt{away} \\ \text{sticky}(x, n) & \text{if } \mathit{name} = \texttt{sticky} \\ \text{nearest}(x, n) & \text{if } \mathit{name} = \texttt{nearest} \\ \text{posinf}(x, n) & \text{if } \mathit{name} = \texttt{posinf} \\ \text{neginf}(x, n) & \text{if } \mathit{name} = \texttt{neginf} \\ \text{trunc}(x, n) & \text{otherwise} \end{cases}$$

5.4 Pseudocode Semantics Revisited

In Subsection 1.1 we defined the semantics of the pseudocode in terms of the function eround.

$$\text{eround}(x, \texttt{[}\mathit{sym}\ \ n\texttt{]}) = \begin{cases} \text{If } x \text{ is not a rational} \\ \quad \text{then err}(\texttt{"non-rational input"}, x) \\ \text{elseif } \mathit{sym} = \texttt{exact} \\ \quad \text{then if } \texttt{[exact } n\texttt{]} \text{ is true of } x \\ \qquad\qquad \text{then } x \text{ else err}(\texttt{"inexact"}, x) \\ \text{elseif } \texttt{[exact } n\texttt{]} \text{ is true of round}(x, \texttt{[}\mathit{sym}\ \ n\texttt{]}) \\ \quad \text{then round(x, } \texttt{[}\mathit{sym}\ \ n\texttt{]}) \\ \quad \text{else err}(\texttt{"bad post-round exponent"}, x) \end{cases}$$

where err(msg, x) returns a list object (i.e., a non-rational "error object") and where we say the *exactness claim* `[exact` n`]`, where n is an integer, $0 < n \leq 64$, is *true* of x if and only if x is an $n,,17$ floating point number.

5.5 The Main Theorem

Main Theorem *If p and d are $64,,15^+$ floating point numbers, $d \neq 0$, and mode is a rounding mode, then* $\text{divide}(p, d, mode) = \text{round}(p/d, mode)$.

Recall that the $64,,15^+$ floating point numbers include all the double extended precision normal and denormal numbers of the IEEE standard.

By definition of "rounding mode", we see that the algorithm supports `trunc`, `away`, `sticky`, `nearest`, `posinf`, and `neginf` rounding, to any precision $0 < n \leq 64$.

6 The Main Proof

To allow clear talk about the values of the variables in the pseudocode we define functions corresponding to the 32 variable names in the pseudocode.

Consider for example line 6

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

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

We call esd_2 the *semantic function* for the pseudocode variable "sd_2." Assuming no previous line signals an error, the meaning of the pseudocode 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 $\mathrm{esd}_2(d)$ may itself signal the first error. Thus, it is somewhat awkward to reason about $\mathrm{esd}_2(d)$ because it is not always numeric.

Now consider the function

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

This can be thought of as the "non-erroneous" value of sd_2. In particular, $\mathrm{esd}_2(d) = \mathrm{sd}_2(d)$ unless the former is an error object. Furthermore, the former is an error object if and only if the precision or exponent of the latter is too big. Thus, not only are the non-erroneous 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.

A "non-erroneous version" of the algorithm is shown in Figure 4, which actually defines all 32 of the non-erroneous semantic functions. Note that we use divide! as the name of the last function, which is the *non-erroneous version* of the division algorithm.

In Figure 4 we follow a convention that we follow henceforth in this paper, namely, when we use a pseudocode variable (other than p, d and $mode$) it is merely an abbreviation for the application of the corresponding non-erroneous semantic function to the appropriate subsequence of p, d, and $mode$. For example, consider line 11 of Figure 4: $q_0 = \mathrm{q}_0(p, d) = \mathrm{away}(sd_2 \times ph_0, 24) = \mathrm{away}(\mathrm{sd}_2(d) \times \mathrm{ph}_0(p), 24)$.

We will prove that the non-erroneous 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* $\mathrm{divide!}(p, d, mode) = \mathrm{round}(p/d, mode)$.

The $64, , 15^+$ hypotheses are necessary only to relate the non-erroneous algorithm to the actual one. We will also prove:

Theorem 2 *If p and d are $64, , 15^+$ floating point numbers, $d \neq 0$ and mode is a rounding mode, then* $\mathrm{divide!}(p, d, mode) = \mathrm{divide}(p, d, mode)$.

Theorems 1 and 2 together imply the main theorem. We prove Theorem 1 in Section 8 and Theorem 2 in Section 9.

1.	$\mathrm{sd}_0(d)$	$= \mathrm{lookup}(d)$
2.	$\mathrm{d}_r(d)$	$= \mathrm{away}(d, 32)$
3.	$\mathrm{sdd}_0(d)$	$= \mathrm{away}(sd_0 \times d_r, 32)$
4.	$\mathrm{sd}_1(d)$	$= \mathrm{trunc}(sd_0 \times \mathrm{comp}(sdd_0, 32), 32)$
5.	$\mathrm{sdd}_1(d)$	$= \mathrm{away}(sd_1 \times d_r, 32)$
6.	$\mathrm{sd}_2(d)$	$= \mathrm{trunc}(sd_1 \times \mathrm{comp}(sdd_1, 32), 32)$
7.	$\mathrm{dh}(d)$	$= \mathrm{trunc}(d, 32)$
8.	$\mathrm{dl}(d)$	$= d - dh$
9.	$\mathrm{p}_0(p)$	$= p$
10.	$\mathrm{ph}_0(p)$	$= \mathrm{trunc}(p_0, 32)$
11.	$\mathrm{q}_0(p, d)$	$= \mathrm{away}(sd_2 \times ph_0, 24)$
12.	$\mathrm{qdh}_0(p, d)$	$= q_0 \times dh$
13.	$\mathrm{qdl}_0(p, d)$	$= q_0 \times dl$
14.	$\mathrm{pt}_1(p, d)$	$= p_0 - qdh_0$
15.	$\mathrm{p}_1(p, d)$	$= pt_1 - qdl_0$
16.	$\mathrm{ph}_1(p, d)$	$= \mathrm{trunc}(p_1, 32)$
17.	$\mathrm{q}_1(p, d)$	$= \mathrm{away}(sd_2 \times ph_1, 24)$
18.	$\mathrm{qdh}_1(p, d)$	$= q_1 \times dh$
19.	$\mathrm{qdl}_1(p, d)$	$= q_1 \times dl$
20.	$\mathrm{pt}_2(p, d)$	$= p_1 - qdh_1$
21.	$\mathrm{p}_2(p, d)$	$= pt_2 - qdl_1$
22.	$\mathrm{ph}_2(p, d)$	$= \mathrm{trunc}(p_2, 32)$
23.	$\mathrm{q}_2(p, d)$	$= \mathrm{away}(sd_2 \times ph_2, 24)$
24.	$\mathrm{qdh}_2(p, d)$	$= q_2 \times dh$
25.	$\mathrm{qdl}_2(p, d)$	$= q_2 \times dl$
26.	$\mathrm{pt}_3(p, d)$	$= p_2 - qdh_2$
27.	$\mathrm{p}_3(p, d)$	$= pt_3 - qdl_2$
28.	$\mathrm{ph}_3(p, d)$	$= \mathrm{trunc}(p_3, 32)$
29.	$\mathrm{q}_3(p, d)$	$= \mathrm{trunc}(sd_2 \times ph_3, 24)$
30.	$\mathrm{qq}_2(p, d)$	$= \mathrm{sticky}(q_2 + q_3, 64)$
31.	$\mathrm{qq}_1(p, d)$	$= \mathrm{sticky}(qq_2 + q_1, 64)$
32.	$\mathrm{divide!}(p, d, mode)$	$= \mathrm{round}(qq_1 + q_0, mode)$

Figure 4: The Non-Erroneous Semantic Functions

7 Fundamental Elementary Results

In our presentation of the proof we take for granted many key properties of the three rounding procedures, trunc, away, and sticky. But in the development of the proof itself, and especially in its mechanization, a large percentage of the total manpower went into the identification and proof of these key properties. We show only a few here.

In this section we implicitly assume that x and y are rationals and i is a positive integer. Recall that the significand of x is $\mathrm{s}(x)$ which we sometimes write as s_x. The exponent of x is $\mathrm{e}(x)$ which may be written e_x.

7.1 Key Properties of the Representation

Lemma 7.1.1 $\mathrm{s}(-x) = \mathrm{s}_x$ *and* $\mathrm{e}(-x) = \mathrm{e}_x$.

Lemma 7.1.2 *If j is an integer,* $\mathrm{s}(x \times 2^j) = \mathrm{s}_x$ *and* $\mathrm{e}(x \times 2^j) = \mathrm{e}_x + j$.

Lemma 7.1.3 *If $x \neq 0$ and j is an integer and $|x| < 2^j$ then* $\mathrm{e}_x < j$.

Lemma 7.1.4 *If $x \neq 0$ and j is an integer and $2^j \leq |x|$ then $j \leq$* e_x.

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

Lemma 7.1.5 *If $x \neq 0$ and $x + y \neq 0$ and* $\mathrm{e}_y \leq \mathrm{e}_x$ *then* $\mathrm{e}(x + y) \leq 1 + \mathrm{e}_x$.

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 7.1.6 *If $x \neq 0$ and $y \neq 0$ and* $\mathrm{e}_y + 1 < \mathrm{e}_x$ *then* $\mathrm{e}_y < \mathrm{e}(x + y)$.

Bounds on the exponent of a product are given by:

Lemma 7.1.7 *If $x \neq 0$ and $y \neq 0$ then* $\mathrm{e}_x + \mathrm{e}_y \leq \mathrm{e}(x \times y) \leq \mathrm{e}_x + \mathrm{e}_y + 1$.

7.2 Elementary Properties of Rounding

Lemma 7.2.1 $\mathrm{trunc}(-x, i) = -\mathrm{trunc}(x, i)$.

Analogous lemmas hold for away and sticky.

Lemma 7.2.2 *For every x there is an ε of the same sign such that* $\mathrm{trunc}(x, i) = x - \varepsilon$ *and* $|\varepsilon| < 2^{\mathrm{e}_x - i + 1}$.

By "same sign" here we mean that if $x < 0$ then $\varepsilon \leq 0$ and otherwise $0 \leq \varepsilon$. Similar lemmas hold for away and sticky:

Lemma 7.2.3 *For every x there is an ε of the same sign such that* $\mathrm{away}(x, i) = x + \varepsilon$ *and* $|\varepsilon| < 2^{\mathrm{e}_x - i + 1}$.

Lemma 7.2.4 *For every x there is an ε such that* $\mathrm{sticky}(x, i) = x + \varepsilon$ *and* $|\varepsilon| < 2^{\mathrm{e}_x - i + 1}$.

Away and sticky have monotonicity lemmas analogous to

Lemma 7.2.5 *If $x \leq y$ then* $\mathrm{trunc}(x, i) \leq \mathrm{trunc}(y, i)$.

How exponents are affected by the rounding procedures are described by

Lemma 7.2.6 *If $x \neq 0$ then* $\mathrm{e}(trunc(x, i)) = \mathrm{e}_x$.

Lemma 7.2.7 *If $x \neq 0$ then*

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

Lemma 7.2.8 *If $x \neq 0$ then* $\mathrm{e}(sticky(x, i)) = \mathrm{e}_x$.

The rounding procedures distribute over multiplication by a power of two:

Lemma 7.2.9 *If j is an integer then* $\mathrm{trunc}(x \times 2^j, i) = \mathrm{trunc}(x, i) \times 2^j$.

Analogous lemmas hold for away and sticky.

The rounding procedures interact simply in certain situations.

Lemma 7.2.10 *If j is an integer and $i \leq j$ then* $\mathrm{trunc}(\mathrm{trunc}(x,i),j) = \mathrm{trunc}(x,i)$.

Lemma 7.2.11 *If j is an integer and $i \leq j$ then* $\mathrm{trunc}(\mathrm{away}(x,i),j) = \mathrm{away}(x,i)$.

Since we use the test "$\mathrm{trunc}(x,n) = x$" to formalize "x fits in n bits" it is Lemma 7.2.11 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 modes.

The informal statement that the product of an m bit number and an n bit number fits in $m+n$ bits is

Lemma 7.2.12 *If m and n are positive integers and* $\mathrm{trunc}(x,m) = x$ *and* $\mathrm{trunc}(y,n) = y$ *then* $\mathrm{trunc}(x \times y, m+n) = x \times y$.

The following two lemmas are especially useful in proving exactness claims.

Lemma 7.2.13 *For every rational x and positive integer i there is an integer j such that* $\mathrm{trunc}(x,i) = j \times 2^{\mathrm{e}_x - i + 1}$, *where $\sigma_j = \sigma_x$ and $2^{i-1} \leq |j| < 2^i$.*

Lemma 7.2.14 *If n is an integer and $|n| < 2^i$, then* $\mathrm{trunc}(n,i) = n$.

7.3 Special Properties of Sticky Rounding

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

Lemma 7.3.1 *If mode is a rounding mode of the form* `[`*name n*`]` *where $n \leq i$ then* $\mathrm{round}(\mathrm{sticky}(x, i+2), \mathit{mode}) = \mathrm{round}(x, \mathit{mode})$.

An especially important fundamental result is

Lemma 7.3.2 (Sticky Plus) *Let x be a non-0 rational that fits in $n > 0$ bits, which is to say* $\mathrm{trunc}(x, n) = x$. *Let y be a rational whose exponent is at least two smaller than that of x, $1 + \mathrm{e}_y < \mathrm{e}_x$. Let k be a positive integer such that $n + \mathrm{e}_y - \mathrm{e}_x < k$.*

$$\overbrace{x\,\overbrace{xxxxxxxxxxxxxxx}^{\mathrm{e}_x}.xxxxxxxxxx}^{n\ bits}$$

$$\underbrace{y\,\underbrace{yyyyyyyyyyy}_{\mathrm{e}_y}.yyyyyyyyyyyyyy\,yyy\ldots}_{k\ bits}$$

Then $\mathrm{sticky}(x + y, n) = \mathrm{sticky}(x + \mathrm{sticky}(y, k), n)$.

The need for Lemma 7.3.2, which is henceforth called the "Sticky Plus" property, 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 Lemma 7.3.2 to be singularly difficult to prove. Our proof considers the signs and relative magnitudes of x and y (intuitively, consider the case that the $+$ sign in the conclusion above is "really" a $-$ sign). Even after that case split, however, the proof is quite interesting. We do not discuss it further here.

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

Lemma 7.3.3 *Let x be a non-0 rational such that* $\mathrm{trunc}(x, n) = x$, *where $n > 1$. Let ε_1 and ε_2 be non-0 rationals such that $|\varepsilon_1| < 2^{\mathrm{e}_x - n + 1}$ and $|\varepsilon_2| < 2^{\mathrm{e}_x - n + 1}$. Furthermore, suppose both ε_1 and ε_2 are positive if either is (i.e., $0 < \varepsilon_1 \leftrightarrow 0 < \varepsilon_2$). Then* $\mathrm{sticky}(x + \varepsilon_1, n) = \mathrm{sticky}(x + \varepsilon_2, n)$.

Despite the fact that we have only stated and not proved the lemmas in this section, it is important for the reader to understand that their proofs were checked mechanically.

8 Proof of Theorem 1

Recall that pseudocode variables here denote calls of the functions of Figure 4.

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

The following proof is a "journal level" description of the one we checked with ACL2.

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

The first six lines of the algorithm of Figure 4 compute an approximation to the reciprocal of d. In Subsection 8.1 we will prove

Lemma 8.1.1 *For every non-0 rational d there exists a rational $0 \leq \varepsilon_{sd2} < 2^{-28}$ such that $sd_2 = (1/d)(1 - \varepsilon_{sd2})$.*

This lemma will enable us 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 8.2.1 (Digit Separation (q_0 *v.* q_1)**)** *If p and d are rationals, $d \neq 0$, and $q_1 \neq 0$ then* $\text{e}(q_1) \leq \text{e}(q_0) - 23$.

The Digit Separation lemma (page 27) 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 non-0 quotient digits have a non-0 sum. For example, if q_2 and q_3 are non-0 then $q_2 + q_3$ is non-0, for otherwise the exponents of q_2 and q_3 would be equal, since $\text{e}(-q_3) = \text{e}(q_3)$. We use these and similar observations implicitly below.

Lines 7 through 9 of the pseudocode 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 non-erroneous 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

$$\begin{aligned} p/d &= p_0/d \\ &= (q_0 + q_1 + q_2 + q_3'). \end{aligned} \qquad (1)$$

Equation (1) tells us 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 $q_3 = \mathrm{trunc}(sd_2 \times \mathrm{trunc}(p_3, 32), 24)$, which is generally different.

In this paper we address only the case where all four quotient digits are non-0 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.

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

However, by Lemma 7.3.1 (page 20) we know

$$divide! \;=\; \mathrm{round}(\psi, mode) \qquad (2)$$

where

$$\psi \;=\; \mathrm{sticky}(q_0 + \mathrm{sticky}(q_1 + \mathrm{sticky}(q_2 + q_3, 64), 64), 66) \qquad (3)$$

We will show that

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

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

The reduction of (4) repeatedly applies Sticky Plus, starting on the inside and working out, appealing to Digit Separation and our non-0 sum observations to relieve the hypotheses:

$$\psi \;=\; \mathrm{sticky}(q_0 + \mathrm{sticky}(q_1 + \mathrm{sticky}(q_2 + \mathrm{sticky}(q_3, 2), 24), 45), 66)$$

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

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

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

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

This is justified because sticky$(q_3, 2)$ and sticky$(q_3', 2)$ satisfy the hypotheses on ε_1 and ε_2 of Lemma 7.3.3 (page 21). In particular, Digit Separation implies $0 < |\text{sticky}(q_3, 2)| < 2^{\text{e}(q_2)-23}$ and $0 < |\text{sticky}(q_3', 2)| < 2^{\text{e}(q_2)-23}$. 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:

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

Thus, we have

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

The last step above is by Lemma 7.3.1 (page 20) and the definition of rounding mode (which insures that the result is rounded to 64 digits or less). **Q.E.D.**

8.1 The Reciprocal Computation

Lemma 8.1.1 *For every non-0 rational d there exists a rational $0 \le \varepsilon_{sd2} < 2^{-28}$ such that $sd_2 = (1/d)(1 - \varepsilon_{sd2})$.*

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 of Figure 4.

Our proof of Lemma 8.1.1 is based on the observation that without loss of generality we can restrict our attention to the case where $1 \le d < 2$. To make this formal, we first observe

Lemma 8.1.2 *If d is a rational and $d \ne 0$ then $\mathrm{sd}_2(d) = \sigma(d) \times \mathrm{sd}_2(\mathrm{s}(d)) \times 2^{-\mathrm{e}(d)}$.*

Proof.

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

Q.E.D.

Given Lemma 8.1.2 it is easy to prove that $\mathrm{sd}_2(\mathrm{s}(d))$ approximates $1/\mathrm{s}(d)$ with the same relative error that $\mathrm{sd}_2(d)$ approximates $1/d$,

Lemma 8.1.3 *If $d \ne 0$ and $\mathrm{sd}_2(\mathrm{s}(d)) = (1/\mathrm{s}(d))(1 - \varepsilon)$, then $\mathrm{sd}_2(d) = (1/d)(1 - \varepsilon)$.*

Hence, we can prove Lemma 8.1.1 by instantiation of Lemma 8.1.4, below: replace d by $\mathrm{s}(d)$; appeal to the fact that for $d \ne 0$, $1 \le \mathrm{s}(d) < 2$; and use Lemma 8.1.3.

Lemma 8.1.4 *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 $k \leq x < k + 2^{-7}$, 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, $\mathrm{truncn}(d, 8)$, then it follows from the monotonicity of multiplication and $k \leq d < k + 2^{-7}$ that $|dv - 1| < \varepsilon$. Thus, if a table is ε-ok and it contains a value v for $\mathrm{truncn}(d, 8)$ 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 $1 \leq d < 2$ (e.g., the 128 8-bit non-0 significands). Hence, by the definition of lookup and the ε-ok property of the table, $|d \times \mathrm{lookup}(d) - 1| < 3/512$.

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

We now move on to lines 2 through 6 of the pseudocode. Observe that if $0 \leq x < 2$, then $\mathrm{trunc}(x, 32) = x(1 - \tau_x)$, for some $0 \leq \tau_x < 2^{-31}$, and $\mathrm{away}(x, 32) = x(1 + \alpha_x)$, for some $0 \leq \alpha_x < 2^{-31}$. 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 $0 \leq \varepsilon_{sd2}(d) < 2^{-28}$ and hence Lemma 8.1.4 and hence Lemma 8.1.1 have both been proved. **Q.E.D.**

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 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 executes the predicate on Table 1 to confirm that it has the required property. (This computation takes about 0.033 seconds on a Sparc Station 20.) Thus, the only time

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

Table 2: Error Analysis for Lines 1-6 ($\delta = 2^{-29}+2^{-31}+(9/512)2^{-31}$)

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.

8.2 Digit Separation

Lemma 8.2.1 (Digit Separation) *Suppose that p and d are rationals and $d \neq 0$. Let $q_3' = p_3/d$. Then*

$$\begin{array}{rcl} q_1 \neq 0 \;\rightarrow\; \mathrm{e}(q_1) & \leq & \mathrm{e}(q_0)-23, \\ q_2 \neq 0 \;\rightarrow\; \mathrm{e}(q_2) & \leq & \mathrm{e}(q_1)-23, \\ q_3 \neq 0 \;\rightarrow\; \mathrm{e}(q_3) & < & \mathrm{e}(q_2)-23, \textit{ and} \\ q_3' \neq 0 \;\rightarrow\; \mathrm{e}(q_3') & < & \mathrm{e}(q_2)-23. \end{array}$$

Proof. In this paper we will prove only the first of the four implications above, namely $q_1 \neq 0 \rightarrow \mathrm{e}(q_1) \leq \mathrm{e}(q_0)-23$. The others are analogous. The relevant lines of code for the first implication are lines 10 through 17 of Figure 4.

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

$$\mathrm{e}(q_1) \;\leq\; \mathrm{e}(q_0)-23$$

is equivalent to

$$\mathrm{e}(\mathrm{away}(sd_2 \times ph_1, 24)) \;\leq\; \mathrm{e}(q_0)-23 \tag{7}$$

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

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

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

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

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

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

But, as we will show, α and β have different signs and their absolute values are bounded strictly above by $2^{\mathrm{e}(q_0)-23}$. 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., $((\alpha \leq 0 \wedge 0 \leq \beta) \vee (\beta \leq 0 \wedge 0 \leq \alpha))$. 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 (i)–(iv).

Lemma 8.2.2 *If x, y and z are rationals then $x - y$ and $y - z$ have different signs if*

$$\begin{array}{lll} (i) & |y| \leq |x| & \textit{and} \\ (ii) & |y| \leq |z| & \textit{and either} \\ (iii) & 0 < x \wedge 0 \leq y \wedge 0 \leq z & \textit{or} \\ (iv) & x \leq 0 \wedge y \leq 0 \wedge z \leq 0 & \end{array}$$

Under the instantiation of x, y, and z above, condition (i) becomes $|sd_2 \times d \times ph_0| \le |p_0|$. But we know $0 < sd_2 \times d \le 1$ by Lemma 8.1.1 (page 25) which tells us that sd_2 approximates $1/d$ from below. Since $|ph_0| = |\mathrm{trunc}(p_0, 32)| \le |p_0|$, condition (i) is proved.

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

Finally we must show either condition (iii) or (iv), which just split on whether x is positive. Here we handle only the case that $0 < x$, i.e., (iii). 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 $d \times \mathrm{away}(sd_2 \times ph_0, 32)$ 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 $|\alpha| < 2^{\mathrm{e}(q_0)-23}$ and $|\beta| < 2^{\mathrm{e}(q_0)-23}$. We address the second first because it is simpler.

Recalling the definition of β from page 28, we wish to prove $|sd_2 \times d(sd_2 \times ph_0 - q_0)| < 2^{\mathrm{e}(q_0)-23}$. Since $0 < sd_2 \times d \le 1$ it suffices to show

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

Expanding the definition of q_0 gives $|sd_2 \times ph_0 - \mathrm{away}(sd_2 \times ph_0, 24)| < 2^{\mathrm{e}(q_0)-23}$. But this follows from $|x - \mathrm{away}(x, i)| < 2^{\mathrm{e}(away(x,i))-i+1}$, which is easily proved from Lemma 7.2.3 (page 19) together with Lemma 7.2.7 (page 19).

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

$$|1/d| \times |(p_0 - sd_2 \times d \times ph_0)| \quad < \quad 2^{\mathrm{e}(q_0)-24}. \tag{12}$$

But we can show

Lemma 8.2.3 $|p_0 - sd_2 \times d \times ph_0| < 2^{\mathrm{e}(p_0)-26}$

Lemma 8.2.4 *If* $p_0 \ne 0$ *then* $|1/d| \times 2^{\mathrm{e}(p_0)} \le 2^{\mathrm{e}(q_0)+2}$

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. **Q.E.D.**

The proofs of Lemmas 8.2.3 and 8.2.4 are left for the reader. Hint: use Lemma 8.1.1 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.

9 Proof of Theorem 2

Theorem 2 *If p and d are $64,,15^+$ floating point numbers, $d \neq 0$ and mode is a rounding mode, then* $\text{divide!}(p, d, mode) = \text{divide}(p, d, mode)$.

9.1 The Non-Erroneous Equivalence Lemmas

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

Lemma 9.1.1 *If d is a $64,,15^+$ floating point number and $d \neq 0$ then* $\text{sd}_2(d) = \text{esd}_2(d)$.

We call this theorem the "non-erroneous equivalence lemma" for sd_2. If we prove such a lemma, e.g., $v = ev$, for each pseudocode variable v (adding appropriate hypotheses for p, d and $mode$ when necessary) then we will have proved Theorem 2 because it is just the corresponding lemma for the last line of code, i.e., divide! is the non-erroneous semantic function corresponding to divide.

But, from the definition of eround, and the non-erroneous equivalence lemmas preceding that for v, we know that $v = ev$ if v is an $n,,17$ floating point number. So we are merely obliged to prove that each v 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.

9.2 Precision

The precision analysis is interesting only for those lines of pseudocode containing an exactness claim. (Lines containing a rounding mode are trivial to handle because the corresponding non-erroneous function is defined to round to the desired precision.) So consider, say, line 15 of Figure 1, where we must prove that p_1 fits in 64 bits, which is to say $\mathrm{trunc}(p_1, 64) = p_1$. Recall here we are dealing with the non-erroneous 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 9.2.1 *If p, d, and q are non-0 rationals such that* $\mathrm{trunc}(p, 64) = p$, $\mathrm{trunc}(d, 64) = d$, $\mathrm{trunc}(q, 24) = q$, *and* $|p - q \times d| \leq |d| \times 2^{\mathrm{e}(q)-23}$, *then* $\mathrm{trunc}(p - q \times d, 64) = p - q \times d$.

We leave the proof to the reader. Hint: consider whether $q \times d$ fits in 87 bits and use Lemmas 7.2.13 and 7.2.14. We also remind the reader that we have mechanically checked the proofs of all the lemmas used in our proof.

9.3 Exponents

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

Our proof considers sequentially the non-erroneous interpretation of each pseudocode variable v and bounds $\mathrm{e}(v)$ in terms of e_p and e_d.

Given the work we did for Lemma 8.1.1 (page 25) the first six lines are straightforward. For example, it is easy to show $-\mathrm{e}_d - 2 \leq \mathrm{e}(sd_2) \leq -\mathrm{e}_d + 1$. Hence, if $-62 - 2^{14} \leq \mathrm{e}_d \leq 2^{14}$, then it is easy to show $1 - 2^{16} \leq \mathrm{e}(sd_2) \leq 2^{16}$.

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

Lemma 9.3.1 *If x and y are rationals such that $x + y \neq 0$, n and m are positive integers,* $\mathrm{trunc}(x, n) = x$ *and* $\mathrm{trunc}(y, m) = y$, *then* $1 - \max(n, m) + \min(\mathrm{e}_x, \mathrm{e}_y) \leq \mathrm{e}(x + y)$.

Lemma 9.3.2 *If x and y are non-0 rationals whose sum is non-0, then* $\mathrm{e}(x+y) \leq 1 + \max(\mathrm{e}_x, \mathrm{e}_y)$.

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.

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

Table 3: Exponent Bounds for Lines 7 through 32

10 Related Work

For an introduction to floating point arithmetic see [5]. See also Goldberg's discussion in Appendix A of [10]. For a detailed treatment of division, *per se*, see [4]. We discuss two areas of related work in formal verification: formalization of floating point arithmetic and mechanically checked proofs of division algorithms.

Part of ANSI/IEEE-854 [12] is formalized in [9] by P. Miner. Miner casts his formalization in the mechanically supported logic of PVS [3]. A few straightforward lemmas about rounding are shown, such as that truncation produces a number of no greater absolute value. These lemmas have presumably been proved mechanically by the PVS system. However, no mechanically checked proofs of floating point algorithms are presented in [9].

There have been several mechanically checked proofs of the SRT division algorithm reported in the literature. In [2] R. E. 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 E. M. Clarke as well as by Clarke, S. M. German and X. Zhao (private communication). In [13] H. Reuss, M. Srivas, and N. Shankar report on the use of the PVS system to verify that a radix r SRT division algorithm divides.

It is perhaps most telling simply to observe that in none of the SRT work cited above is it necessary to formalize the notions of floating point number or rounding to state or prove the theorems reported. But those concepts are key to the algorithm and theorem discussed here.

11 Concluding Remarks

We have mechanically checked a proof that the kernel of the floating point division algorithm used on the AMD5$_K$86 microprocessor is correct in the sense that on $64,,15^+$ floating point numbers (which includes the double extended precision normal and denormal numbers of the IEEE standard), it returns the $n,,17$ floating point number obtained by rounding the "infinitely precise" quotient by the method and to the precision specified by the rounding mode.

The successful checking of this proof establishes that it is possible to apply an existing general purpose theorem proving tool to some floating point algorithms of practical interest.

References

[1] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams," *ACM Computing Surveys* **40**(3) 293–318, September, 1992

[2] R. E. Bryant, "Bit-Level Analysis of an SRT Divider Circuit," CMU-CS-95-140, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213.

[3] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A Tutorial Introduction to PVS," presented at *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, April 1995 (see http://www.csl.sri.com/pvs.html).

[4] M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer Academic Publishers: Norwell, MA, 1994.

[5] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, **23**(1) 5–48, March, 1991.

[6] M. Kaufmann and J S. Moore, *ACL2 Version 1.8*, URL ftp://ftp.cli.com/-pub/acl2/v1-8/acl2-sources/doc/HTML/acl2-doc.html, 1995.

[7] M. Kaufmann and J S. Moore, "ACL2: An Industrial Strength Version of Nqthm," submitted.

[8] 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, May 26, 1995.

[9] P. M. Miner, "Defining the IEEE-854 Floating-Point Standard in PVS," NASA Technical Memorandum 110167, NASA Langely Research Center, Hampton, VA 23681, 1995.

[10] D. A. Patterson and J. L. Hennessy, *Computer Architecture*, Morgan Kaufmann Publishers: San Mateo, CA., 1990.

[11] Standards Committee of the IEEE Computer Society. *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std. 754-1985, IEEE, 345 East 47th Street, New York, NY 10017, (1985).

[12] Standards Committee of the IEEE Computer Society. *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, IEEE Std. 854-1987, IEEE, 345 East 47th Street, New York, NY 10017, (1987).

[13] H. Ruess, M. K. Srivas, and N. Shankar, "Modular Verification of SRT Division," Computer Science Laboratory, SRI International, Menlo Park, CA 49025, 1996.

[14] G. L. Steele Jr. *Common LISP: The Language*, Digital Press: Bedford, MA, 1984.

[15] G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.