Formal Verification of Divide and Square Root Algorithms using Series Calculation

Jun Sawada
IBM Austin Research Laboratory
Email: sawada@us.ibm.com

Abstract

IBM Power4$^{\text{TM}}$ processor uses series approximation to calculate divide and square root. We formally verified that the algorithms with a series of rigorous error bound analysis using the ACL2 theorem prover. In order to carry out the verification, we need to show that the error of the Chebyshev series used in the square root algorithm is suitably bounded. This was performed by analyzing the Chebyshev series using Taylor series. In general, Taylor series has less accuracy than Chebyshev series, however, we used ACL2's macro extensions and computed hints to automatically generate Taylor serious approximation for small segments, and proved that the approximation error in each of these segments is smaller than the requirement.

1 Introduction

In this paper, we discuss the formal verification of the divide and square root algorithms used in the IBM Power4$^{\text{TM}}$ processor. This algorithm was first proven, not formally, by Agarwal et al. in [AGS99]. Conventional mathematical analysis, such as that given in the work of Agarwal, provides a great insight into algorithms and a certain level of assurance. However, hand-proof generally does not provide an absolute assurance on the correctness. Mechanical theorem proving, on the other hand, provides a greater confidence in the proven algorithm by systematically checking every detail of the algorithm.

There are a number of formal verification studies on the floating-point unit of industrial microprocessors. Moore et al. used the ACL2 theorem prover [KM96] to verify the microcode for the divide algorithm used in the AMD-K5 processor [MLK98]. Russinoff later verified the microcode for the square root algorithms in the same processor [Rus99]. Our work is very similar to these works, as we verify the divide and square root algorithms that are encoded as microcodes of the Power4 processor. We also find divide and square root algorithm verification in the later work by Russinoff [Rus98] and Aagaard et al. [AJK^{+}00], which pay more attention to fill the "semantic gap" between the low level implementation with gates and wires and the high level mathematical specifications.

One thing that distinguishes our work from others is the error bound calculation for the series approximation used in the algorithm. The divide and square root algorithms verified in the work mentioned above use the Newton-Raphson algorithm [PH96]. This algorithm, starting from an initial estimate, calculates a better estimate from a previous one. The formula to obtain a new estimate is relatively simple, and it takes a few iterations to obtain an accurate enough estimate which is rounded to the final answer according to a specified rounding mode. In the Newton-Raphson algorithm, many instructions are dependent on earlier instructions. The algorithm may require more execution cycles on a processor with many pipeline stages and high latency.

IBM Power4 processor and its predecessor Power3$^{\mathrm{TM}}$ processor use a different method of function iteration for divide and square root. From the initial approximation, it obtains a better approximation using series approximation. Series calculation needs more instructions than a single iteration of the Newton-Raphson algorithm. However, only one iteration is sufficient to obtain the necessary precision. Since instructions in the series calculation are less dependent on earlier instructions than those in the Newton-Raphson algorithm, more instructions can be executed in parallel with a pipelined floating-point unit.

One challenge in the formal verification is the calculation of the error by the series approximation. Specifically, the square root algorithm uses the Chebyshev series, which provides a better approximation than the Taylor series of the same degree. We want to verify that this series has errors small enough to guarantee that the final estimate is rounded to the correct answer.

In order to formally prove the error size of the Chebyshev series, we defined Taylor series and admitted Taylor's theorem as an axiom in ACL2. Then we used it to calculate the error of the Chebyshev series. This allowed us to avoid the complex mathematics of Chebyshev series. With the help of ACL2 macros and computed-hint features, we automatically generated hundreds of Taylor series to give an accurate analysis of the Chebyshev series.

2 Divide and Square Root Algorithm

In this section, we explain the divide and square root algorithm used in the Power4 processor. Both algorithms use a table-lookup to obtain an initial estimate of the answer, derive a better estimate using series calculation, and round it to the final answer.

Let us first discuss the divide algorithm. The mantissa part and the exponent part are independently calculated. The mantissa part calculation is essential to the algorithm, as the exponent calculation is simply a subtraction and an adjustment based on the mantissa result. Thus, we assume that $1 \le a, b < 2$, and discuss the algorithm to calculate a/b. A reference algorithm to calculate the division for the entire range has been written, but we did not model the logic for exponent calculation and rounding implemented in the Power4 processor.

First, we introduce a few functions. We define $\mathrm{expo}(x)$ as the function that returns the exponent of x. Function $\mathrm{ulp}(x, n)$ returns the unit of least position that is defined as:

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

This corresponds to the magnitude of the least significant bit of an n-bit precision floating point number x. Function $\mathrm{near}(x, n)$ rounds a rational number x to the floating-point number with n-bit precision using IEEE nearest mode rounding. Function $\mathrm{rnd}(x, m, n)$ rounds x to n-bit precision floating-point number using rounding mode m, where m can be *near*, *trunc*, *inf* or *minf* corresponding

Table 1: Double-precision floating-point divide algorithms used in Power4.

Algorithm to calculate a/b
Look up y_0 $e := \mathrm{near}(1 - y_0 \times b, 53)$ $q_0 := \mathrm{near}(y_0 \times a, 53)$ $t_1 := \mathrm{near}(1/2 + e \times e, 53)$ $y_1 := \mathrm{near}(y_0 + y_0 \times e, 53)$ $e_2 := \mathrm{near}(a - b \times q_0, 53)$ $t_2 := \mathrm{near}(3/4 + t_1 \times t_1, 53)$ $t_3 := \mathrm{near}(y_1 \times e_2, 53)$ $q_1 := q_0 + t_2 \times t_3$ div-round(q_1,a, b, mode)

to the four IEEE rounding modes [Ins]. These functions, except $\mathrm{ulp}(x, n)$, are defined in the ACL2 public library.

The division algorithm obtains an initial estimate of $1/b$ from a table on the chip, which will be called y_0. The initial estimate is accurate enough to satisfy $|1 - y_0 \times b| < 1/256$. Let $e = 1 - y_0 \times b$. An estimate of a/b is given by the product $q_0 = y_0 \times a$. The error of the estimate q_0 can be calculated as follows:

$$a/b - q_0 \quad = a/b \times (1 - by_0) = a/b \times e$$

By solving this equation with respect to a/b, we can get $a/b = q_0/(1 - e) \simeq q_0(1 + e + e^2 + e^3 + e^4 + e^5 + e^6)$. This equation is used to calculate a better estimate of a/b from the initial estimate q_0. The actual algorithm is given in Table 1.

This divide algorithm calculates a better estimate q_1 from q_0 using a series of multiply-and-add instructions. Instructions are ordered so that only a few of them are truly-dependent on immediately preceding instructions. This allows the pipelined floating-point unit to execute multiple instructions in parallel.

Function div-round(q_1, a, b, m) at the end of the algorithm rounds the final estimate q_1 to the correct answer $\mathrm{rnd}(a/b, m, 53)$. This rounding involves a multiplication, a comparison and an adjustment. For example, when $m = \mathit{inf}$ and $b \times \mathit{near}(q_1, 53) < a$, the function returns $\mathit{near}(q_1, 53) + \mathit{ulp}(q_1, 53)$ as the rounded result. This rounding is performed by an internal instruction in Power4 processor, and it works only under the condition $|a/b - q_1| < \mathrm{ulp}(q_1, 53)/4$. This is the condition that must be verified for the final estimate q_1 in the divide algorithm.

The square root algorithm is similar to the divide algorithm. Again, the calculation of exponent is not essential for the algorithm, thus we describe the algorithm to calculate the mantissa of the square root of b, assuming that $1/2 \leq b < 2$.

We look-up two initial approximations from on-chip tables. One is for the 12-bit estimate of $\sqrt{b}$, which will be named as q_0. Another is 53-bit rounded approximation of $1/q_0^2$, which is called y_0. The tables are given only for $1/2 \leq b < 1$ to reduce the size of the on-chip tables. For $1 \leq b < 2$, we lookup the table for $b/2$ and adjust the values by dividing y_0 by 2 and multiplying q_0 by `*root2*`, which is a precomputed 53-bit representation of $\sqrt{2}$. The adjusted

Table 2: Double-precision floating-point square root algorithms used in Power4.

Algorithm to calculate $\sqrt{b}$
Look up y_{0s}
Look up q_0
$e :=$ near$(1 - y_{0s} \times b, 53)$
$q_{0s} :=$ near$(*root2* \times q_0, 53)$ if $1 \leq b < 2$
$:= q_0$ if $1/2 \leq b < 1$
$t_3 :=$ near$(c_4 + c_5 \times e, 53)$
$t_4 :=$ near$(c_2 + c_3 \times e, 53)$
$esq :=$ near$(e \times e, 53)$
$t_5 :=$ near$(c_0 + c_1 \times e, 53)$
$e_1 :=$ near$(q_{0s} \times q_{0s} - b, 53)$
$t_1 :=$ near$(y_{0s} \times q_{0s}, 53)$
$t_6 :=$ near$(t_4 + esq \times t_3, 53)$
$q_{0e} :=$ near$(t_1 \times e_1, 53)$
$t_7 :=$ near$(t_5 + esq \times t_6, 53)$
$q_1 := q_{0s} + q_{0e} \times t_7$
sqrt-round(q_1,b,mode)

values are called y_{0s} and q_{0s}, respectively. Estimate y_{0s} is close enough to $1/b$ to satisfy $|e| \leq 2^{-6}$, where $e = 1 - y_{0s}b$.

The difference between the squares of q_{0s} and $\sqrt{b}$ can be calculated as:

$$q_{0s}^2 - b \simeq q_{0s}^2(1 - b \times y_{0s}) = q_{0s}^2 e$$

By solving this equation with respect to $\sqrt{b}$, we get

$$\sqrt{b} \simeq q_{0s}\sqrt{1-e} \simeq q_{0s}(1 + c_0 e + c_1 e^2 + c_2 e^3 + c_3 e^4 + c_4 e^5 + c_5 e^6)$$

where $1 + c_0 e + \cdots + c_5 e^6$ is the Chebyshev series approximation of $\sqrt{1-e}$. Further manipulation of the right-hand side will lead to:

$$\begin{aligned}
\sqrt{b} &\simeq q_{0s} + q_{0s}e(c_0 + \cdots c_5 e^5) \\
&= q_{0s} + q_{0s}(1 - y_{0s}b)(c_0 + \cdots c_5 e^5) \\
&\simeq q_{0s} + q_{0s}(q_{0s}^2 y_{0s} - y_{0s}b)(c_0 + \cdots c_5 e^5) \\
&= q_{0s} + q_{0s}y_{0s}(q_{0s}^2 - b)(c_0 + \cdots c_5 e^5)
\end{aligned}$$

The algorithm in Table 2 uses this equation to calculate a better approximation q_1 of $\sqrt{b}$. The procedure to obtain y_{0s} from y_0 is not explicit in Table 2 as it is simply an exponent adjustment. Chebyshev coefficients c_0 through c_5 are 53-bit precision floating-point numbers obtained from an on-chip table. In fact, we use two sets of Chebyshev coefficients, one of which is intended to be used for $0 \leq e \leq 2^{-6}$ and the other for $-2^{-6} \leq e < 0$. Let c_{0p}, c_{1p}, c_{2p}, c_{3p}, c_{4p} and c_{5p} be the first set of coefficients intended for the positive case, and c_{0n}, c_{1n}, c_{2n}, c_{3n}, c_{4n} and c_{5n} be for the negative case. In our algorithm, the 6th fraction bit of b, instead of the polarity of e, determines which set of coefficients will be used. This can be justified by the fact that e tends to be positive when the 6th fraction bit of b is 0, and negative otherwise. However, this relation between the 6th fraction bit of b and the polarity of e is not always true, and we must verify that this heuristic in selecting Chebyshev coefficients does not cause too much error. The analysis in Section 4 takes care of this problem.

The function sqrt-round(q_1, b, m) at the end of the algorithm rounds the final estimate q_1 to the correct answer $\text{rnd}(\sqrt{b}, m, 53)$, under the condition $|q_1 - \sqrt{b}| < \text{ulp}(q_1, 53)/4$. The trick to implement this function can be found in [AGS99], together with that of the rounding function for divide.

Summarizing the algorithm description so far, both divide and square root algorithms look up an initial estimate from the table, calculate a better estimate q_1 with the error of less than a quarter of ulp, and round it to determine the final answer. Our verification objective is to prove q_1 falls into this required error margin.

3 Verification Overview

In this section, we provide an overview of the verification proof of the divide and square root algorithm. The proof is basically the same as that presented by Agarwal et al.[AGS99] Here, we will focus on the way the proof is formalized by the ACL2 theorem prover. In the formal proof, we are not allowed to simplify formulae using approximations, such as $\sqrt{1+x} \simeq 1 + x/2$ for a very small x, as was done in Agarwal's paper. Every formula transformation must be exact, and this will eliminate the ambiguity of the hand-proof. Every small error is formally defined and analyzed, and it is the only way to carry out the formal proof using a mechanical theorem prover.

3.1 Proof of Divide

First, we define intermediate values for the divide algorithm during the calculation of a/b. We define $\tilde{y_0}$, $\tilde{e}$, $\tilde{q_0}$, $\tilde{t_1}$, $\tilde{y_1}$, $\tilde{e_2}$, $\tilde{t_2}$ and $\tilde{t_3}$ to be the values before rounding during the calculation of y_0, e, q_0, t_1, y_1, e_2, t_2, and t_3, respectively. We also define r_{y_0}, r_e, r_{q_0}, r_{t_1}, r_{y_1}, r_{e_2}, r_{t_2}, and r_{t_3} as the amount added to the raw values by rounding. More formally, the definition can be given as:

$$\begin{array}{lll}
\tilde{e} = 1 - b \times y_0 & e = \text{near}(\tilde{e}, 53) & r_e = e - \tilde{e} \\
\tilde{q_0} = a \times y_0 & q_0 = \text{near}(\tilde{q_0}, 53) & r_{q_0} = q_0 - \tilde{q_0} \\
\tilde{t_1} = 1/2 + e \times e & t_1 = \text{near}(\tilde{t_1}, 53) & r_{t_1} = t_1 - \tilde{t_1} \\
\tilde{y_1} = y_0 + y_0 \times e & y_1 = \text{near}(\tilde{y_1}, 53) & r_{y_1} = y_1 - \tilde{y_1} \\
\tilde{e_2} = a - b \times q_0 & e_2 = \text{near}(\tilde{e_2}, 53) & r_{e_2} = e_2 - \tilde{e_2} \\
\tilde{t_2} = 3/4 + t_1 \times t_1 & t_2 = \text{near}(\tilde{t_2}, 53) & r_{t_2} = t_2 - \tilde{t_2} \\
\tilde{t_3} = y_1 \times e_2 & t_3 = \text{near}(\tilde{t_3}, 53) & r_{t_3} = t_3 - \tilde{t_3}
\end{array}$$

In fact, these values are defined as functions of a and b in the ACL2 formalization, but we omit the arguments for presentation purposes.

By automatic case-analysis of the look-up table for y_0, ACL2 shows that $|\tilde{e}| \leq 2^{-8}$ and also $|e| \leq 2^{-8}$. The amount rounded off by the nearest-mode rounding is at most half of the ulp as stated in the following lemma.

Lemma 1 *For rational number x and a positive integer n,*

$$|near(x, n) - x| \leq ulp(x, n)/2$$

From this lemma, we can show that $|r_e| \leq 2^{-61}$. The magnitude of other intermediate values can be similarly calculated as:

$|\tilde{q_0}| < 2$	$|q_0| \leq 2$	$|r_{q_0}| < 2^{-53}$
$|\tilde{t_1}| < 1$	$|t_1| \leq 1$	$|r_{t_1}| < 2^{-54}$
$|\tilde{y_1}| < 1$	$|y_1| \leq 1$	$|r_{y_1}| < 2^{-54}$
$|\tilde{e_2}| < 3/2 \times 2^{-7}$	$|e_2| < 2^{-6}$	$|r_{e_2}| < 2^{-60}$
$|\tilde{t_2}| < 2$	$|t_2| < 2$	$|r_{t_2}| < 2^{-53}$
$|\tilde{t_3}| < 2^{-6}$	$|t_3| \leq 2^{-6}$	$|r_{t_3}| < 2^{-60}$

Next, we represent each intermediate value as the sum of a formula the intermediate value is intended to represent and an error term. For example, t_1 is the sum of $1/2 + \tilde{e} \times \tilde{e}$ and its error term $E_{t_1} = 2r_e\tilde{e} + r_e^2 + r_{t_1}$.

$$\begin{aligned} t_1 &= 1/2 + (\tilde{e} + r_e) \times (\tilde{e} + r_e) + r_{t_1} \\ &= 1/2 + \tilde{e} \times \tilde{e} + 2r_e\tilde{e} + r_e^2 + r_{t_1} \\ &= 1/2 + \tilde{e} \times \tilde{e} + E_{t_1}, \end{aligned}$$

From the magnitude of the intermediate values, the size of the error term E_{t_1} can be calculated as:

$$|E_{t_1}| < 2|r_e||\tilde{e}| + |r_e|^2 + |r_{t_1}| < 2^{-54} + 2^{-67}$$

Similarly, with appropriate error terms E_{y_1}, E_{t_2}, E_{t_3}, and E_{q_1} whose definitions are given in the appendix, we can represent y_1, t_2, t_3 and q_1 in the following way:

$y_1 = y_0 + y_0\tilde{e} + E_{y_1}$	$|E_{y_1}| < 2^{-54} + 2^{-61}$
$t_2 = 1 + \tilde{e}^2 + \tilde{e}^4 + E_{t_2}$	$|E_{t_2}| < 2^{-52}$
$t_3 = y_0(1 + \tilde{e})\tilde{e_2} + E_{t_3}$	$|E_{t_3}| < 2^{-58}$
$q_1 = q_0 + y_0\tilde{e_2}(1 + \tilde{e} + \tilde{e}^2 + \tilde{e}^3 + \tilde{e}^4 + \tilde{e}^5) + E_{q_1}$	$|E_{q_1}| < 2^{57} - 2^{-101}$

The right-hand side of the last equation can be further manipulated as follows:

$$\begin{aligned} q_1 &= q_0 + y_0\tilde{e_2}\frac{1 - \tilde{e}^6}{1 - \tilde{e}} + E_{q_1} \\ &= q_0 + y_0(a - bq_0)\frac{1 - \tilde{e}^6}{by_0} + E_{q_1} \\ &= q_0 + (a/b - q_0) - (a/b - \tilde{q_0} - r_{q_0})\tilde{e}^6 + E_{q_1} \end{aligned}$$

Since $a/b - \tilde{q_0} = a/b(1 - by_0) = a/b \times \tilde{e}$, we can further simplify the formula to

$$q_1 = a/b(1 - \tilde{e}^7) + r_{q_0}\tilde{e}^6 + E_{q_1} = a/b(1 - \tilde{e}^7) + E_{final} \tag{1}$$

where we define E_{final} to be $r_{q_0}\tilde{e}^6 + E_{q_1}$. We can easily calculate $|E_{final}| < 2^{-57}$.

Finally, we show that the $|q_1 - a/b|$ is less than $\text{ulp}(q_1, 53)/4$ by case analysis. Suppose $q_1 \geq 1$. Since $a/b < 2$, we get using the equation (1):

$$|q_1 - a/b| \leq |a/b||\tilde{e}|^7 + |E_{final}| < 2^{-55} + 2^{-57} < 2^{-54} = \text{ulp}(q_1, 53)/4$$

Suppose $q_1 < 1$. Then, again from the equation (1),

$$a/b = \frac{q_1 - E_{final}}{1 - \tilde{e}^7} < \frac{1 + 2^{-57}}{1 - 2^{-56}}$$

Therefore,

$$|q_1 - a/b| \leq |a/b||\tilde{e}|^7 + |E_{final}| < 2^{-55} = \mathrm{ulp}(q_1, 53)/4.$$

Either way, $|q_1 - a/b|$ is less than $\mathrm{ulp}(q_1, 53)/4$.

Summarizing the proof, it was carried out in the following steps:

1. Define the intermediate values, and prove the upper bound of the absolute values.

2. Calculate the deviation of intermediate values from the approximated formula.

3. Using the results from the previous steps, calculate the deviation of the final estimate from the infinitely precise result.

Mechanization of the proof using ACL2 was straightforward. We encoded each step as ACL2 theorems, and then verified the theorems in ACL2. Sometimes we needed to fill the logical gap between theorems to aid the mechanical proof, but the ACL2 theorem prover, once proper libraries had been loaded and fine-tuned, was good at simplifying formulae and calculating the upper bound for error terms.

3.2 Proof of Square Root

In the proof of the square root algorithm, we took an approach which was similar to the one for the divide algorithm. However, there are a few important differences to note. First, we used ACL2(r) [Gam99] to carry out the proof. ACL2(r) is a version of ACL2 which allows analysis on real numbers using non-standard analysis[Rob59]. Square root can return irrational numbers, and we can only define and reason about it directly using ACL2(r). The other major difference is we need to calculate the approximation error by Taylor series and Chebyshev series, which will be discussed in detail in the next section.

First we define intermediate values $q_{0s}, y_{0s}, e, t_3, t_4, esq, t_5, e_1, t_1, t_6, q_{0e}, t_7$ and q_1. These are, in fact, functions of b, but we omit the argument b for simplicity in the paper. The same is true for the Chebyshev coefficients $c0$, $c1$, $c2$, $c3$, $c4$ and $c5$, which are selected from two sets of coefficients depending on b. For each of the intermediate values, we define $\tilde{e}$, $\tilde{t_3}$, $\tilde{t_4}$, $\tilde{esq}$, $\tilde{t_5}$, $\tilde{e_1}$, $\tilde{t_1}$, $\tilde{t_6}$, $\tilde{q_{0e}}$ and $\tilde{t_7}$ as the infinitely precise value before rounding. We define r_e, r_{t_3}, r_{t_4}, r_{esq}, r_{t_5}, r_{e_1}, r_{t_1}, r_{t_6}, $r_{q_{0e}}$ and r_{t_7} as the values added to the infinitely precise values by rounding. Additionally, we define μ as $\mu = y_{0s}q_{0s}^2 - 1$.

Analyzing the look-up tables for y_{0s} and q_{0s} using ACL2(r), we can easily show that $|e| < 2^{-6}$, $|\mu| \leq 397/128 \times 2^{-53}$, $50/71 \leq q_{0s} < 71/50$ and $1/2 \leq y_{0s} < 2$. We calculate the magnitude of intermediate values and the adjusting values for rounding. Some of the upper bounds of intermediate values are $|\tilde{esq}| \leq 2^{-12}, |\tilde{t_7}| \leq 2^{-1} + 2^{-6}, |\tilde{e_1}| \leq 2^{-5} + 2^{-50}$ and $|\tilde{q_{0e}}| \leq 183/128 \times 2^{-5}$.

Next, by defining appropriate error terms $E_{q_{0e}}$, E_{esq}, E_{t_3}, E_{t_4}, E_{t_5}, E_{t_6} and E_{t_7}, we represent intermediate values as the sum of the formula it intends to represent and an error term.

$$\begin{array}{ll}
q_{0e} = q_{0s}(\tilde{e}+\mu) + E_{q0e} & |E_{q_{0e}}| \leq 2^{-56} \\
esq = \tilde{e}^2 + E_{esq} & |E_{esq}| \leq 2^{-64} \\
t_3 = c_4 + c_5\tilde{e} + E_{t_3} & |E_{t_3}| \leq 2^{-58} + 2^{-65} \\
t_4 = c_2 + c_3\tilde{e} + E_{t_4} & |E_{t_4}| \leq 2^{-57} + 2^{-64} \\
t_5 = c_0 + c_1\tilde{e} + E_{t_5} & |E_{t_5}| \leq 2^{-54} + 2^{-61} \\
t_6 = c_2 + c_3\tilde{e} + c_4\tilde{e}^2 + c_5\tilde{e}^3 + E_{t_6} & |E_{t_6}| \leq 2^{-56} + 2^{-63} \\
t_7 = c_0 + c_1\tilde{e} + c_2\tilde{e}^2 + c_3\tilde{e}^3 + c_4\tilde{e}^4 + c_5\tilde{e}^5 + E_{t_7} & |E_{t_7}| \leq 2^{-53} + 2^{-60}
\end{array}$$

Let $P(x)$ denote the polynomial[1] $c_0 + c_1x + c_2x^2 + c_3x^3 + c_4x^4 + c_5x^5$. Then

$$q_1 = q_{0s} + q_{0s}(\tilde{e}+\mu)P(\tilde{e}) + E_{q1} \qquad |E_{q1}| \leq 2^{-56}$$

for some appropriate error term E_{q1}.

We are going to rewrite the last equation using a number of series approximation. We define

$$\begin{aligned}
E_{su} &= \sqrt{1+\mu} - (1+\mu/2) \\
E_{se} &= \sqrt{1-\tilde{e}} - (1-\tilde{e}/2) \\
E_{cheb} &= \sqrt{1-\tilde{e}} - (1+\tilde{e}P(\tilde{e}))
\end{aligned}$$

Further we define the following error terms:

$$\begin{aligned}
E_{pet2} &= P(\tilde{e}) - (-1/2 - \tilde{e}/8) \\
E_{sb} &= q_{0s} \times E_{se} - \sqrt{b} \times (E_{su} + \mu/2) \\
E_{final} &= -3/8 \times q_{0s}\mu\tilde{e} - q_{0s}E_{cheb} + \mu E_{sb}/2 + \sqrt{b}E_{su} + E_{q1} + q_{0s}\mu E_{pet2}
\end{aligned}$$

Then we can prove that

$$q_1 = \sqrt{b} + E_{final}$$

The details of the proof are provided in the appendix.

As we discuss in the following section, we can prove that:

$$\begin{aligned}
|E_{su}| &\leq 2^{-105} \\
|E_{se}| &\leq 2^{-15} + 2^{-19} \\
|E_{cheb}| &\leq 3/2 \times 2^{-58}
\end{aligned}$$

From these inequalities and the definition of E_{final}, we can prove:

$$|E_{final}| < 2^{-55} \leq \text{ulp}(b, 53)/4$$

Thus the new estimate q_1 is less than one quarter of the ulp away from $\sqrt{b}$.

4 Use of Taylor's Theorem in Error Size Calculation

In the proof of the square root algorithm in the previous section, we skipped the proof of the upper bounds of $|E_{su}|, |E_{se}|$ and $|E_{cheb}|$. In this section, we introduce Taylor series and use it in their proof.

[1] Since coefficients c_0 through c_5 depend on b, $P(x)$ depends on b as well. In ACL2(r), we define it as a function that takes b and x as its arguments.

First, we introduce $\sqrt{x}$ as an ACL2(r) function `(sroot` x`)`, using the ACL2(r) function stub mechanism, which introduces a function without definition. Then, we assumed that `(sroot` x`)` satisfies the following two axioms.

```
(defaxiom realp-sroot
  (implies (and (realp b) (>= b 0))
           (and (realp (sroot b)) (>= (sroot b) 0))))

(defaxiom square-sroot
  (implies (and (realp b) (>= b 0))
           (equal (* (sroot b) (sroot b)) b)))
```

The first theorem states that $\sqrt{b}$ is a non-negative real number, and the second states that $\sqrt{b} \times \sqrt{b} = b$. Because $\sqrt{x}$ is defined as a function without definition, ACL2(r) cannot directly calculate the value of applications of this square root function. This is true even if we define the square root using non-standard analysis, because $\sqrt{x}$ can return an irrational number which cannot be handled directly by ACL2(r).

However, there is a function that approximates square roots. ACL2(r) function `(iter-sqrt` x ϵ`)` in the public library distributed with ACL2 returns a rational number close to $\sqrt{x}$. In this paper, we write $\sqrt{x}^*_\epsilon$ to denote this function. This function satisfies $\sqrt{x}^*_\epsilon \times \sqrt{x}^*_\epsilon \leq x$ and $x - \sqrt{x}^*_\epsilon \times \sqrt{x}^*_\epsilon < \epsilon$ for a positive rational number ϵ. From this, we can easily prove that $\sqrt{x} - \sqrt{x}^*_\epsilon \leq max(\epsilon, \epsilon/x)$.

Taylor series approximation is a simple and frequently used approximation for differentiable functions. Taylor's theorem states

Taylor's Theorem *If $f^{(n)}(x)$ is continuous in $[x_0, x_0+\delta]$ and $f^{(n+1)}(x)$ exists in $(x_0, x_0+\delta)$, then there exists $\xi \in (x_0, x_0+\delta)$ and*

$$\begin{aligned} f(x_0+\delta) &= f(x_0) + f'(x_0)\delta + \frac{f''(x_0)}{2!}\delta^2 + \cdots + \frac{f^{(n-1)}(x_0)}{(n-1)!}\delta^{(n-1)} + \frac{f^{(n)}(\xi)}{n!}\delta^n \\ &= \sum_{i=0}^{n-1} \frac{f^{(i)}(x_0)}{i!}\delta^i + \frac{f^{(n)}(\xi)}{n!}\delta^n \end{aligned}$$

The term $\sum_{i=0}^{n-1} \frac{f^{(i)}(x_0)}{i!}\delta^i$ is often used as an approximation of the function f at point x_0. The error of Taylor series approximation is given by the term $\frac{f^{(n)}(\xi)}{n!}\delta^n$, which is called the Taylor remainder.

Since the square root function is infinitely differentiable and its derivatives are continuous for the positive domain, we can apply Taylor's theorem on the entire positive domain. Let us define $a(n, x)$ as:

$$a(n, x) = \frac{1}{n!}\prod_{i=0}^{n-1}(\frac{1}{2} - i)x^{\frac{1}{2}-n}$$

Function $a(n, x_0)$ gives the n'th Taylor coefficient for $\sqrt{x}$ at x_0. Then Taylor's equation for square root can be given as:

$$\sqrt{x_0+\delta} = \sum_{i=0}^{n-1} a(i, x_0)\delta^i + a(n, \xi)\delta^n$$

Given that (`nth-tseries-sroot` i x_0 δ) represents i'th term in the Taylor series $a(i, x_0)\delta^i$, we define the Taylor series approximation in the ACL2(r) logic as:

```
(defun tseries-sroot (n x delta)
  (if (zp n)
      0
    (+ (nth-tseries-sroot (- n 1) x delta)
       (tseries-sroot (- n 1) x delta))))
```

Since ξ in Taylor's theorem depends on n, x and δ, we represent ξ using ACL2(r) stub function (`taylor-sroot-xi` n x δ). Then Taylor's theorem for square root can be defined as an ACL2(r) axiom.

```
(defaxiom taylor-theorem-on-sroot
  (implies (and (integerp n) (< 0 n)
                (realp x) (< 0 x)
                (realp delta) (< 0 (+ x delta)))
   (equal (sroot (+ x delta))
          (+ (tseries-sroot n x delta)
             (nth-tseries-sroot n (taylor-sroot-xi n x delta)
                                  delta)))))
```

Additionally, we constrain (`taylor-sroot-xi` n x δ) to return a real number that is in the open segment $(x, x + \delta)$.

```
(defaxiom type-taylor-sroot-xi
  (implies (and (integerp n) (< 0 n)
                (realp x) (realp delta) (< 0 x) (< 0 (+ x delta)))
           (realp (taylor-sroot-xi n x delta))))

(defaxiom range-taylor-sroot-xi
  (implies (and (integerp n) (< 0 n)
                (realp x) (realp delta) (< 0 x) (< 0 (+ x delta)))
           (and (<= (min x (+ x delta))
                    (taylor-sroot-xi n x delta))
                (<= (taylor-sroot-xi n x delta)
                    (max x (+ x delta))))))
```

Summarizing so far, we have added a square root function without concrete definition and admitted a few facts about it including Taylor's theorem as axioms. Alternatively, we can use ACL2(r)'s non-standard analysis feature to define square root function, prove the basic properties and Taylor's theorem without adding new axioms. However, in this paper, we are interested in using the square root function in the verification of an algorithm, not the definition of the square root function and proof of its basic properties. So we chose to skip these elaborate definitions and fundamental proofs.

An upper bound of $|E_{su}|$ can be calculated by applying Taylor's theorem. Since E_{su} is equal to the second degree Taylor remainder for the function $\sqrt{1+\mu}$ at $\mu = 0$:

$$E_{su} = \sqrt{1+\mu} - (1 + \mu/2) = -\frac{1}{8}(1+\xi)^{-\frac{3}{2}}\mu^2.$$

Since $|\mu| \leq \frac{397}{128} \times 2^{-53}$ and $|\xi| < |\mu|$, an upper bound of $|E_{su}|$ is given as:

$$|E_{su}| < \frac{1}{8}(1 - \frac{397}{128} \times 2^{-53})^{-\frac{3}{2}} \times (\frac{397}{128} \times 2^{-53})^2 < 2^{-105}$$

Similarly, the upper bound for $|E_{se}|$ can be calculated as:

$$|E_{se}| < \frac{1}{8} \times (1 - 2^{-6})^{-\frac{3}{2}} \times (2^{-6})^2 < 2^{-15} + 2^{-19}$$

We used Taylor series in the calculation of an upper bound of $|E_{cheb}|$ as well. Since the Chebyshev series $1 + \tilde{e}P(\tilde{e})$ is a better approximation of $\sqrt{1 - \tilde{e}}$ than the Taylor series of the same degree, it is not straightforward to use Taylor series in the measurement of its approximation error.

Our approach is dividing the range of $\tilde{e}$ into small segments, generating a Taylor series for each segment and using it to calculate the error of the Chebyshev series for every segment one at a time. Each segment should be small enough so that the generated Taylor series is a far more accurate approximation of the square root function than the Chebyshev series. The range of $\tilde{e}$ is $[-2^{-6}, 2^{-6}]$. We divided it into 128 segments of size 2^{-12}, and performed error analysis on each segment.

Since there are a large number of segments, we must automate the error analysis at each segment. One of the obstacles is that ACL2(r) cannot perform calculation on irrational numbers directly. For example, the Taylor coefficient $a(i, x_0)$ is usually an irrational number. In order to automate the error calculation, we define an ACL2(r) function that calculates the approximation of $a(i, x_0)$ using the function $\sqrt{x}^*_\epsilon$. More precisely,

$$a^*(x, i, \eta) = \frac{1}{n!}\prod_{i=0}^{n-1}(1/2 - i)\sqrt{x}^*_\eta x^{-n}$$

Then we can show:

$$|a(x, n) - a^*(x, n, \eta)| \leq \frac{1}{n!}\prod_{i=0}^{n-1}(1/2 - i) \times max(\eta, \eta/x)x^{-n}$$

As discussed in Section 2, our algorithm selects Chebyshev coefficients from two sets of constants depending on the 6th fraction bit of b. Let

$$\begin{aligned} Cheb_p(e) &= 1 + c_{0p}e + c_{1p}e^2 + c_{2p}e^3 + c_{3p}e^4 + c_{4p}e^5 + c_{5p}e^6 \\ Cheb_n(e) &= 1 + c_{0n}e + c_{1n}e^2 + c_{2n}e^3 + c_{3n}e^4 + c_{4n}e^5 + c_{5n}e^6 \end{aligned}$$

Then $E_{cheb} = \sqrt{1 - \tilde{e}} - Cheb_p(\tilde{e})$ when the 6th fraction bit of b is 0, and $E_{cheb} = \sqrt{1 - \tilde{e}} - Cheb_n(\tilde{e})$ when it is 1.

Let us calculate the size of E_{cheb} for the case where the 6th fraction bit of b is 1. A simple analysis shows that $-2^{-6} \leq \tilde{e} \leq 3/2 \times 2^{-12}$. Note that $\tilde{e}$ can be slightly positive, even though our heuristic suggests that $\tilde{e}$ is negative when the 6th fraction bit of b is 1. We divide this domain of $\tilde{e}$ into 66 segments by substituting $e_0 - e_\delta$ for $\tilde{e}$ in $\sqrt{1 - \tilde{e}} - Cheb_n(\tilde{e})$, where e_0 is one of the 66 constants -63×2^{-12}, -62×2^{-12}, ..., 2×2^{-12}, while e_δ is a new variable that satisfies $0 \leq e_\delta \leq 2^{-12}$. The upper bound for the entire domain of $\tilde{e}$ is simply the maximum value of all the upper bounds for the 66 possible choices for e_0.

The upper bound for $|E_{cheb}|$ can be represented as the summation of three terms.

$$\begin{aligned}|\sqrt{1-e_0+e_\delta} - Cheb_n(e_0-e_\delta)| \quad \leq \quad & \left|\sqrt{1-e_0+e_\delta} - \sum_{i=0}^{5} a(1-e_0,i)e_\delta^i\right| \\ & + \left|\sum_{i=0}^{5} a(1-e_0,i)e_\delta^i - \sum_{i=0}^{5} a^*(1-e_0,i,\eta)e_\delta^i\right| \\ & + \left|\sum_{i=0}^{5} a^*(1-e_0,i,\eta)e_\delta^i - Cheb_n(e_0-e_\delta)\right|\end{aligned}$$

An upper bound for the first term can be given by applying Taylor's theorem.

$$\begin{aligned}|\sqrt{1-e_0+e_\delta} - \sum_{i=0}^{5} a(1-e_0,i)e_\delta^i| \leq |a(\xi,6)e_\delta^6| \leq \frac{1}{6!}\prod_{i=0}^{5}|\frac{1}{2}-i||\xi^{-\frac{11}{2}}||e_\delta^6| \\ < \frac{1}{6!}\prod_{i=0}^{5}(\frac{1}{2}-i) \times max((1-e_0)^{-6},(1-e_0)^{-5}) \times 2^{-78}\end{aligned}$$

Here ξ is the constant satisfying Taylor's theorem such that $1-e_0 < \xi < 1-e_0+e_\delta$. Note that this upper bound can be calculated by ACL2(r) as it does not contain square root nor variables.

The upper bound for the second term can be calculated as follows:

$$\begin{aligned}|\sum_{i=0}^{n-1} a(1-e_0,i)e_\delta^i - \sum_{i=0}^{n-1} a^*(1-e_0,i,\eta)e_\delta^i| \leq \sum_{i=0}^{n-1}|a(1-e_0,i) - a^*(1-e_0,i,\eta)|e_\delta^i \\ \leq \sum_{i=0}^{n-1}\left\{\frac{2^{-13i}}{i!}\prod_{j=0}^{i-1}(1/2-j) \times max(\eta,\eta/(1-e_0)) \times (1-e_0)^{-i}\right\}\end{aligned}$$

We chose η to be 2^{-60} to make this term small enough. Again the upper bound has no variables involved and can be calculated by ACL2(r).

The third term is the difference between the Chebyshev series approximation and the Taylor series approximation. Since e_0 and η are constant in the third term, we can simplify the term $\sum_{i=0}^{5} a^*(1-e_0,i,\eta)e_\delta^i - Cheb_n(e_0-e_\delta)$ into a polynomial of e_δ of degree 6. Here having the computational function $a^*(1-e_0,i,\eta)$ rather than the real Taylor coefficient allows ACL2(r) to automatically simplify the formula. We denote the resulting polynomial as $\sum_{i=0}^{6} b_i e_\delta^i$, where coefficient b_i is a constant automatically calculated by ACL2(r) during the simplification. Then the upper bound can be given as:

$$\left|\sum_{i=0}^{5} a^*(1-e_0,i,\eta)e_\delta^i - Cheb_n(e_0-e_\delta)\right| = |\sum_{i=0}^{6} b_i e_\delta^i| \leq \sum_{i=0}^{6}|b_i| \times 2^{-13i}$$

By adding the three upper bounds, we can prove that

$$|\sqrt{1-e_0+e_\delta} - Cheb_n(e_0-e_\delta)| < 3/2 \times 2^{-58}$$

for all 66 values for e_0. This is the upper bound of $|E_{cheb}|$ when the 6th fraction bit of b is 1. Similarly, we can prove that $|\sqrt{1-e_0+e_\delta} - Cheb_p(e_0-e_\delta)| <$

$3/2 \times 2^{-58}$ for the case where the 6th fraction bit is 0. In this case, $-6/5 \times 2^{-12} \leq \tilde{e} \leq 2^{-6}$. Since the ranges of $\tilde{e}$ are overlapping for the two cases, we repeat the upper bound analysis on some segments. Summarizing the two cases, $|E_{cheb}|$ has the upper bound $3/2 \times 2^{-58}$.

5 Discussions

We have formally verified that the Power4 divide and square root algorithms return the final estimate whose error is less than a quarter of the ulp. The main proof was carried out by defining error terms and analyzing their size at each step of the algorithm. One major challenge for the verification was evaluating the approximation error for Chebyshev series. We have performed error size calculation of the Chebyshev series approximation in hundreds of small segments. For each segment, a Taylor series is generated to evaluate the approximation error of the Chebyshev series. This type of proof can be carried out only with a mechanical theorem prover or other type of computer program, because the simplification of hundreds of formulae is too tedious for humans to carry out correctly.

The upper bound proof of the Chebyshev series approximation was carried out automatically after providing the following:

1. Macros that provide the template of the proof for small segments.
2. Computed hints that spawn the case analysis automatically.
3. A set of rewrite rules that simplify a complex formula into a polynomial of rational coefficients.

Since the proof is automatic, we could change a number of parameters to try different configurations. For example, we changed the segment size and η used to calculate $\sqrt{x}^*_\eta$. In fact, Chebyshev series approximation error was obtained by trial-and-error. At first, we set a relatively large number to an ACL2(r) constant `*apx_error*` and ran the prover to verify $|E_{cheb}| <$`*apx_error*`. If it is successful, we lowered the value of `*apx_error*`, iterated the process until the proof failed. The details of macros and computed hints are discussed in Appendix C.

The approximation error analysis using Taylor series requires less computational power than brute-force point-wise analysis. When $|\tilde{e}| \leq 2^{-6}$, the value $\sqrt{1-\tilde{e}} \simeq 1-\tilde{e}/2$ ranges approximately from $1-2^{-7}$ to $1+2^{-7}$. In order to prove that the error of its Chebyshev series approximation is less than 1.5×2^{-58}, simple calculation suggests that we need to check nearly 2^{50} points. On the other hand, the entire verification of the square root algorithm with approximation calculation on 128 segments took 673 seconds on a Pentium III 400MHz system. It is not a sheer luck that we could finish the error calculation by analyzing only hundreds of segments. Because the size of the n'th degree Taylor remainder for the square root function is $O(d^n)$ for the segment size d, the approximation error by a Taylor series quickly converges to 0 by using high degree Taylor series and making the segment smaller. We believe that we can apply our technique to other algorithms involving series calculations.

For our proof, we assumed the correctness of Taylor's theorem, instead of proving it. However, ACL2(r) can prove Taylor's theorem by carrying out non-standard analysis. In fact, Gamboa and Middleton recently proved Taylor's theorem in ACL2(r)[GM02]. It is our future work to integrate their result into the proof presented in this paper.

There might be a question why we did not assume the known facts about Chebyshev series and use them in the series analysis. One answer is that the mathematics behind Chebyshev series is much more complex than Taylor series. We also need to develop ACL2(r) library to compute the approximation of integration which would be used in the calculation of Chebyshev coefficients. It is also more likely that we might introduce incorrect axioms about Chebyshev series because its definition is complex.

Finally, we must acknowledge the usefulness of the floating-point library in the books distributed with ACL2. This library, originally developed by David Russinoff, supplied most of the basic theorems about floating point numbers and rounding. This allowed us to focus on the verification of the algorithm.

References

[AGS99] Ramesh C. Agarwal, Fred G. Gustavson, and Martin S. Schmookler. Series approximation methods for divide and square root in the power3 processor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 116–123, 1999.

[AJK$^+$00] Mark D. Aagaard, Robert B. Jones, Roope Kaivola, Katherine R. Kohatsu, and Carl-Johan H. Seger. Formal verification of iterative algorithms in microprocessors. *Proceedings Design Automation Conference (DAC 2000)*, pages 201 – 206, 2000.

[Gam99] Ruben Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2.* PhD thesis, University of Texas at Austin, 1999.

[GM02] R. A. Gamboa and B. E. Middleton. Taylor's formula with remainder. In *ACL2 Workshop 2002*, 2002.

[Ins] Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754-1985.

[KM96] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of nqthm. In *Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34. IEEE Computer Society Press, June 1996.

[MLK98] J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the AMD5$_K$86 Floating-Point Division Program. *IEEE Trans. Comp.*, 47(9):913–926, September 1998.

[PH96] David A. Patterson and John L. Hennessey. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition, 1996.

[Rob59] A. Robinson. Model theory and non-standard arithmetic, infinitistic methods. In *Symposium on Foundations of Mathmatics*, 1959.

[Rus98] D. M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division, and Square Root Algorithm of the AMDK7 Processor. *J. Comput. Math. (UK)*, 1, 1998.

[Rus99] D. Russinoff. A Mechanically Checked Proof of Correctness of the AMDK5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, 14(1), 1999.

[Saw00] Jun Sawada. ACL2 computed hints: Extension and practice. In *ACL2 Workshop 2000 Proceedings, Part A*. The University of Texas at Austin, Department of Computer Sciences, Technical Report TR-00-29, November 2000.

A Error Terms for the Divide Proof

$$\begin{aligned}
E_{y_1} &= y_0 r_e + r_{y_1} \\
E_{t_2} &= 2E_{t_1}(1/2 + \tilde{e}^2) + E_{t_1}^2 + r_{t_2} \\
E_{t_3} &= (y_0 + y_0\tilde{e})r_{e_2} + E_{y_1}(\tilde{e_2} + r_{e_2}) + r_{t_3} \\
E_{q_1} &= E_{t_2} y_0 (1 + \tilde{e})\tilde{e_2} + (1 + \tilde{e}^2)\tilde{e}^4 E_{t_3} + E_{t_2} E_{t_3}
\end{aligned}$$

B Proof Detail of Square Root Algorithm

First we provide the error term definitions missing from Section 3.

$$\begin{aligned}
E_{q_{0e}} &= \tilde{t_1} r_{e_1} + r_{t_1} e_1 + r_{q_{0e}} \\
E_{esq} &= 2\tilde{e} r_e + r_e^2 + r_{esq} \\
E_{t_3} &= c_5 r_e + r_{t_3} \\
E_{t_4} &= c_3 r_e + r_{t_4} \\
E_{t_5} &= c_1 r_e + r_{t_5} \\
E_{t_6} &= E_{t_4} + \tilde{e}^2 E_{t_3} + t_3 E_{esq} + r_{t_6} \\
E_{t_7} &= E_{t_5} + \tilde{e}^2 E_{t_6} + t_6 E_{esq} + r_{t_7} \\
E_{q_1} &= E_{q_{0e}} P(\tilde{e}) + q_{0e} E_{t_7}
\end{aligned}$$

Now we discuss the derivation of $q_1 = \sqrt{b} + E_{final}$ from $q_1 = q_{0s} + q_{0s}\tilde{e}P(\tilde{e}) + q_{0s}\mu P(\tilde{e}) + E_{q_1}$.

$$\begin{aligned}
\sqrt{b} &= \sqrt{b(1+\mu)} - \sqrt{b}(E_{su} + \frac{\mu}{2}) \\
&= q_{0s}\sqrt{b y_{0s}} - \sqrt{b}(E_{su} + \frac{\mu}{2}) \\
&= q_{0s}\sqrt{1 - \tilde{e}} - \sqrt{b}(E_{su} + \frac{\mu}{2})
\end{aligned}$$

$$\begin{aligned}
&= q_{0s} - q_{0s}\frac{\tilde{e}}{2} + q_{0s}E_{se} - \sqrt{b}(E_{su} + \frac{\mu}{2}) \\
&= q_{0s} - q_{0s}\frac{\tilde{e}}{2} + E_{sb}
\end{aligned}$$

Using this equation, we can derive:

$$\begin{aligned}
q_{0s} + q_{0s}\tilde{e}P(\tilde{e}) &= q_{0s}(\sqrt{1-\tilde{e}} - E_{cheb}) \\
&= \sqrt{b(1+\mu)} - q_{0s}E_{cheb} \\
&= \sqrt{b} + \sqrt{b}\frac{\mu}{2} + \sqrt{b}E_{su} - q_{0s}E_{cheb} \\
&= \sqrt{b} + \frac{\mu}{2}(q_{0s} - q_{0s}\frac{\tilde{e}}{2} + E_{sb}) + \sqrt{b}E_{su} - q_{0s}E_{cheb}
\end{aligned}$$

From the definition of E_{pet2},

$$q_{0s}\mu P(\tilde{e}) = q_{0s}\mu(-\frac{1}{2} - \frac{\tilde{e}}{8} + E_{pet2})$$

By adding the two equations above:

$$\begin{aligned}
q_1 &= q_{0s} + q_{0s}\tilde{e}P(\tilde{e}) + q_{0s}\mu P(\tilde{e}) + E_{q_1} \\
&= \sqrt{b} - \frac{3}{8}q_{0s}\mu\tilde{e} - q_{0s}E_{cheb} + \frac{1}{2}E_{sb}\mu + \sqrt{b}E_{su} + q_{0s}\mu E_{pet2} + E_{q1} \\
&= \sqrt{b} + E_{final}
\end{aligned}$$

In this appendix, we showed the detail of the equation rewriting. The ACL2 theorem prover can figure out the detail with appropriate hints, thus we do not have to feed the detailed proof into the prover. For example, the proof shown here was proven with 5 `defthm`'s.

C Use of Macros and Computed Hints for Square Root Proof

We describe the macros and computed hints used in the verification of the square root algorithm. The macro shown in Figure 1 generates the ACL2 proof of the upper bound of $|E_{cheb}|$ discussed in Section 4. It has three parameters: `n`, `lemma-namebase` and `apx-fun`. This macro analyzes the upper bound of $|E_{cheb}|$ when e ranges over the segment [`n` $\times$ `*apx-epsilon*`, (`n` $+ 1) \times$ `*apx-epsilon*`], where `*apx-epsilon*` is an ACL2(r) constant equal to 2^{-12}. Parameter `lemma--namebase` provides the name of the main lemma, and `apx-fun` selects either $Cheb_p$ or $Cheb_n$ as the function analyzed in the proof.

This macro generates an ACL2(r) encapsulate expression which contains four local lemmas and one exported lemma. The first three local lemmas prove the upper bound for the three terms from Section 4 whose summation gives the upper bound of $|\sqrt{1 - e_0 + e_\delta} - Cheb_n(e_0 - e_\delta)|$ (when $Cheb_n$ is passed for the argument `apx-fun`). The fourth local lemma combines these lemmas and gives a constant upper bound for $|\sqrt{1 - e_0 + e_\delta} - Cheb_n(e_0 - e_\delta)|$. The last main lemma rephrases the result of the fourth lemma by substituting e for $e_0 - e_\delta$.

There are several parameters found in the macro: `*apx-epsilon*` specifies the segment size to carry out the upper bound analysis at a time, `*apx-eta*` defines η used to calculate the function $a^*(x,i,\eta)$, `*tc-difference*` specifies the upper bound for the third lemma, and `*apx-error*` is the upper bound for $|E_{cheb}|$. We can easily change the value of these parameters and rerun the proof to find a better upper bound.

Another macro `prove-all-apx-error-bound-lemmas` makes it easier to repeatedly perform the upper bound proof for the small segments. For example,

```
(prove-all-apx-error-bound-lemmas -63 2 abs-E_cheb-neg-case- Cheb-n)
```

calls `(apx-error-bound-lemmas` n `abs-E_cheb-neg-case- Cheb-n)` 66 times by varying n from -63 to 2.

We combined the lemmas for the 66 segments into a single lemma. The proof of the combined lemma goes like this way: first we case-split the target formula into 67 subgoals that cover all the segments, then we apply the proper lemma to each subgoal by `:use` hint. The next defthm implements this approach using computed-hints.

```
(defthm abs-E_cheb-neg
  (implies (and (rationalp e)
                (<= (- (expt 2 -6)) e)
                (<= e (* 3/2 (expt 2 -12))))
           (<= (abs (- (sroot (- 1 e)) (Cheb-n e)))
               *apx-error*))
  :hints ((case-split-i-=-x-to-y '(FL  (* (expt 2 12) e)) (- (expt 2 6)) 1)
          (when-GS-match-& ((0) (*) . 0)
                           (gen-use-apx-neg-lemma-ajusted-index))))
```

The computed-hint `(case-split-i-=-x-to-y '`*expr* n m`)` splits the goal into $m-n+1$ cases where *expr* ranges over the integers between n and m. It is defined as:

```
(defun collect-for (i x y)
  (declare (xargs :measure (nfix (- (1+ y) x))))
  (if (and (integerp x) (integerp y) (<= x y))
      (cons `(equal ,i ,x) (collect-for i (1+ x) y))
    nil))

(defun case-split-i-=-x-to-y (i x y)
  `(:cases ,(collect-for i x y)))
```

The computed-hint `(when-GS-match-& pat more-hint)` tries another computed hint `more-hint` when the goal spec matches the pattern provided as `pat` [Saw00]. In the `defthm` above, it applies the computed hint `gen-use--apx-neg-lemma-ajusted-index` on every immediate subgoal after the case split. The computed hint `gen-use-apx-neg-lemma-ajusted-index` selects the proper lemma out of 66 candidates, and insert it by a `:use` hint. It is defined as:

```
(defmacro gen-use-apx-neg-lemma-ajusted-index ()
  `(let ((idx (caadr id)))
     (if (and (<= 1 idx) (<= idx 66))
         (gen-use-lemma-with-suffix 'abs-E_cheb-neg-case-
                                    (+ 3 (- idx)))
       nil)))
```

```
(defmacro apx-error-bound-lemmas (n lemma-namebase  apx-fun)
  (let* ( <Variables used for Lemma Name Generation>
         (name-of-lemma-1  <Name of Lemma 1>)
         (name-of-lemma-2  <Name of Lemma 2>)
         (name-of-lemma-3  <Name of Lemma 3>)
         (name-of-lemma-4  <Name of Lemma 4>)
         (name-of-lemma    <Main Lemma Name>))
   `(encapsulate nil
      (local
       (defthm ,name-of-lemma-1
         (implies (and (rationalp ed) (equal e0  (* ,n *apx-epsilon*))
                       (<= 0 ed) (<= ed *apx-epsilon*))
                  (<= (abs (- (sroot (+ 1 (- e0) ed))
                              (tseries-sroot 6 (+ 1 (- e0)) ed)))
                      (taylor-rem-sr-ub-2 6 (- 1 e0) *apx-epsilon*)))
          <Lemma Options>))
      (local
       (defthm ,name-of-lemma-2
         (implies (and (rationalp ed) (equal e0  (* ,n *apx-epsilon*))
                       (<= 0 ed) (<= ed *apx-epsilon*))
                  (<= (abs (- (tseries-sroot 6 (+ 1 (- e0)) ed)
                              (p-tseries-sroot 6 (+ 1 (- e0)) ed *apx-eta*)))
                      (E_p-tseries-sroot 6 (+ 1 (- e0))
                                         *apx-epsilon* *apx-eta*)))
         <Lemma Options>))
      (local
       (defthm ,name-of-lemma-3
         (implies (and (rationalp ed) (equal e0  (* ,n *apx-epsilon*))
                       (<= 0 ed) (<= ed *apx-epsilon*))
                  (<= (abs (- (p-tseries-sroot 6 (+ 1 (- e0)) ed *apx-eta*)
                              (,apx-fun (- e0 ed))))
                      *tc-difference*))
         <Lemma Options>))
      (local
       (defthm ,name-of-lemma-4
         (implies (and (rationalp ed) (rationalp e0)
                       (equal e0 (* ,n *apx-epsilon*))
                       (<= 0 ed) (<= ed *apx-epsilon*))
                  (<= (abs (- (sroot (- 1 (- e0 ed)))
                              (,apx-fun (- e0 ed))))
                      *apx-error*))
         <Lemma Options>))
      (defthm ,name-of-lemma
        (implies (and (rationalp e)
                      (<= (* (- ,n 1) *apx-epsilon*) e)
                      (<= e (* ,n *apx-epsilon*)))
                 (<= (abs (- (sroot (- 1 e)) (,apx-fun e)))
                     *apx-error*))
        <Lemma Options>)))) ; end of proof macro
```

Figure 1: Macro to generate lemmas for the error analysis of $|E_{cheb}|$. Some details have been cut out and replaced with an italic text in angles.

A computed-hint receives the current goal spec through the variable `id`. This information is sufficient for the computed-hint to construct the name of the proper lemma to be applied.

The macros and computed hints allowed us to succinctly write the theorems. It makes it easier to perform the proof with various parameters. In this sense, the macros and computed-hints were essential to carry out the verification work presented in this paper.