Verification of IEEE Compliant Subtractive Division Algorithms

Paul S. Miner[1] and James F. Leathrum, Jr.[2]

[1] NASA Langley Research Center, Hampton VA 23681-0001 USA,
E-mail: p.s.miner@larc.nasa.gov
[2] Old Dominion University, Department of Electrical and Computer Engineering,
Norfolk VA 23529 USA, E-mail: leathrum@ee.odu.edu

Abstract. A parameterized definition of subtractive floating point division algorithms is presented and verified using PVS. The general algorithm is proven to satisfy a formal definition of an IEEE standard for floating point arithmetic. The utility of the general specification is illustrated using a number of different instances of the general algorithm.

1 Introduction

As computing systems become more complex, it becomes increasingly difficult to ensure that testing fully exercises the design. This was made abundantly clear by the infamous bug in the floating point unit of the Intel Pentium(TM) microprocessor. The bug consists of five missing entries in a lookup table. Pratt [20] provides a thorough analysis of this error. He provides compelling arguments that a thorough manual analysis of a design may still allow errors to evade detection. This is particularly true if the flaw is in a region of the design that is thought to be unreachable. Machine assisted reasoning is crucial to prevent such errors. The SRT divider [19] verified by Rueß, Srivas, and Shankar [17] illustrates how the Prototype Verification System (PVS) [16] can be used to prevent omissions in lookup tables similar to that employed by the Pentium (TM). Their work describes a formal relationship from a verified algorithm to a formal description of a hardware design. In order to complete the verification, it is necessary to relate the algorithm to a formal description of the floating point operation.

Our work provides a formal relationship from the IEEE floating point standards [6, 7] to a class of verified division algorithms. A strength of theorem prover based verification is that it allows verification of classes of algorithms. Once a class is verified with respect to the standard, it is trivial to instantiate it with a specific instance. The SRT divider [19] verified in [17] is an instance of the class we have verified. Other instances include the classical restoring and nonrestoring division algorithms. Thus, our general theory provides a standard specification for IEEE compliant subtractive division.

The verification is structured in a hierarchical fashion. At the root is a verification of the class of subtractive division algorithms. We illustrate the utility of the general verification by exhibiting a number of instances. Then it is shown how these algorithms can be extended to provide IEEE compliant rounding.

These instances provide a collection of formal specifications for implementations of floating point division that are known to meet the standard.

2 Related work

Much of the previous work in theorem prover based verification of floating point algorithms has focused on verifying core algorithms and a corresponding hardware design. The first efforts targeted verified implementation of binary nonrestoring algorithms. Leeser, O'Leary, and others present a verification (using *Nuprl*) of a binary nonrestoring square root algorithm and its implementation [12, 9]. Verkest, et al present a similar verification (using *nqthm*) of a binary nonrestoring division algorithm [21].

In response to the flaw in the Pentium [20], several researchers investigated theorem prover based verifications of SRT division hardware. Clarke, German, and Zhao used the ANALYTICA theorem prover to verify Taylor's [19] radix-4 SRT division circuit [4]. Their verification includes an abstract representation of the lookup table and a proof that it defines all necessary values for the quotient selection logic. Rueß, Srivas, and Shankar generalize this work using the PVS theorem prover. They present a general verification of arbitrary radix SRT division algorithms, instantiate their theory with Taylor's radix-4 SRT division circuit [19], and verify a description of the hardware. Included in their work is a technique to verify a concrete representation of the lookup table. By virtue of PVS' type system, proof obligations are automatically generated to ensure that blank entries in the table are inaccessible.

In all of the efforts above, the verifications address the functional correctness of floating point algorithms and an associated hardware design. They do not address the issue of relating the algorithms to a formal definition of floating point operations. Harrison has presented a verification of two floating point algorithms; square root, and a CORDIC [22] natural logarithm algorithm [8]. For both algorithms, Harrison relates the proofs to a floating point interpretation. Although he does not present hardware descriptions, he does address some of the preliminary error analysis necessary to provide correct rounding. The IEEE standards for floating point arithmetic unambiguously state that operations

> *shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then that result rounded according to one of the modes ...* [6, 7]

Barrett manually verified a general rounding algorithm with respect to a Z formalization of IEEE 754 [1, 2]. When Barrett performed his verification, there was no machine assisted reasoning for the Z specification language. Some tools for machine assisted application of Z have recently been developed [13]. Thus far, these have not been applied to floating point verification.

Recently, the microcode for the floating point division and square root algorithms of the AMD5$_K$86^{TM} microprocessor has been mechanically verified using the *ACL2* theorem prover [15, 18]. Both algorithms assume correct hardware for

floating point multiplication, addition, and subtraction. Both verifications include detailed analysis of rounding and proof that the delivered result is rounded in accordance with the IEEE standard. In addition, the verifications guarantee that all intermediate results of the algorithms fit the datapath of the existing floating point hardware.

Our work is a generalization of the Rueß, Srivas, and Shankar verification. In addition, to the SRT algorithms, our verification encompasses most of the algorithms presented in [5]. In addition, our verification includes a formal path relating the algorithm to the IEEE standard.

2.1 Brief introduction to PVS

The Prototype Verification System (PVS) [16] is a verification system that provides support for general purpose theorem proving. The specification language is a higher order logic augmented with dependent types. Theories can be parameterized, and the dependent type mechanism allows for stating arbitrary constraints on theory parameters. The type system of PVS includes predicate subtypes and is therefore undecidable. PVS frequently generates proof obligations to ensure that expressions are well typed. PVS has powerful decision procedures, so many proofs involving simple arithmetic expressions can be discharged automatically. In addition, PVS provides a collection of pre-proven results in the `prelude`. Also included with PVS are libraries providing support for bit-vectors and finite sets. In PVS, the real numbers are a base type, and other numeric types are defined as subtypes of the reals. This allows specifications to freely mix operations on numeric types.

3 General verification of subtractive division algorithms

There are two principle classes of floating point division algorithms. The subtractive algorithms use shifting followed by addition/subtraction to generate quotient digits in time linear with respect to the size of the operands. Multiplicative algorithms, such as Goldschmidt's Algorithm or Newton-Raphson iterations provide fewer iterations, but the operations in each iteration grow increasingly complex. Ercegovac and Lang present a detailed study of subtractive algorithms algorithms for both division and square root [5]. The general division algorithm is presented in PVS providing a parameterized class of verified subtractive division algorithms.

3.1 General algorithm definition

Subtractive algorithms generate one quotient digit per iteration. They are designed so that in iteration i the remainder is no larger than r^{-i} for a radix-r algorithm. Ercegovac and Lang present a series of interrelated factors which differentiate subtractive division algorithms: the radix (r), the quotient-digit set ($\{-a, -a+1, ..., -1, 0, 1, ..., a-1, a\}$), the range of the divisor (b), and the

quotient-digit select function (qs) [5]. These factors are all parameters to the general division theory. The formal parameters are:

```
    r  : {i : posint | i > 1},
    a  : {i : posint | ceiling((r-1)/2) <= i & i < r},
    b  : {i : posint | 1 < i & i <= r},
 (IMPORTING divide_types[r, a, b])
   qs  : qs_type
```

The first three parameters are defined using predicate subtypes of the positive integers. In addition, the types for a and b are constrained by the value of r. The type signature for function qs depends on all of the previous parameters. The declaration for type *qs_type* is imported from theory *divide_types*. Theory *divide_types* declares constant $\rho = \frac{a}{r-1}$ and the following types:

```
    D_type    : TYPE = {d : real     | 1 <= d & d < b}
    dividend  : TYPE = {x : posreal | 1/r <= x & x < rho}
    p_type(D) : TYPE = {p : real     | abs(p) <= rho*D}
    qs_type   : TYPE =
        [D : D_type, p : p_type(D) ->
         {q : subrange(-a,a) | abs(r*p - q*D) <= rho*D}]
```

Constant ρ denotes the redundancy factor of the quotient-digit set. It represents a trade-off in design complexity between the quotient selection function and generation of divisor multiples. The type of the divisor is constrained by *D_type*, which is defined to include the numeric range of the significand of a normalized floating point number. The type of the dividend is constrained to ensure that it satisfies constraints imposed on the partial remainder. Parameterized type *p_type(D)* encodes an invariant, dependent on divisor D, that the partial remainder must satisfy during execution of the algorithm. Finally, the type of the quotient selection function, *qs_type*, is restricted to functions that, given a divisor D and a partial remainder p such that $|p| \leq \rho \cdot D$, return a digit q such that $-a \leq q \leq a$ and $|r \cdot p - q \cdot D| \leq \rho \cdot D$.

The subtractive division algorithms, given divisor D and dividend X, are characterized by the following recurrence equations for the quotient q and the partial remainder p:

$$\begin{aligned} q_i &= r \cdot q_{i-1} + qd_i \\ p_i &= r \cdot p_{i-1} - qd_i \cdot D \end{aligned} \tag{1}$$

where $q_0 = 0$, $p_0 = X$, and qd_i is the quotient digit selected for iteration i. A PVS specification of Equation 1 is:

```
  divide(X,D)(n) : RECURSIVE [# p : p_type(D), q: integer #] =
    IF n=0 THEN (# p := X, q := 0 #)
           ELSE (# p := r*p(divide(X,D)(n-1)) - qd_n*D,
                   q := r*q(divide(X,D)(n-1)) + qd_n #)
                WHERE qd_n = qs(D,p(divide(X,D)(n-1))) ENDIF
    MEASURE n
```

By using record types for the range of the function, the definition is a direct transliteration of the recurrence equations. In addition, by declaring the partial remainder to be of type *p_type(D)*, PVS automatically generates a proof obligation to ensure that the invariant is satisfied. This obligation is proven using the type constraints on the quotient selection function.

3.2 Verification of the general algorithm

To simplify later definitions we define the abbreviations:

Definition 1.

$$p(X,D)(n) \hat{=} p(\mathit{divide}(X,D)(n))$$
$$q(X,D)(n) \hat{=} q(\mathit{divide}(X,D)(n))$$

PVS strategy (induct-and-simplify) proves:

Lemma 2. $p(X,D)(n) \cdot r^{-n} = X - q(X,D)(n) \cdot r^{-n} \cdot D$

The invariant property of the partial remainder guarantees

Lemma 3. $\left|\dfrac{p(X,D)(n) \cdot r^{-n}}{D}\right| \leq r^{-n}$

To strengthen the convergence property, the algorithm includes a corrective step after the final iteration.

Definition 4.

$$P(X,D)(n) \hat{=} \begin{cases} (p(X,D)(n)+D)\cdot r^{-n} & \text{if } p(X,D)(n) < 0 \\ (p(X,D)(n)-D)\cdot r^{-n} & \text{if } p(X,D)(n) = D \\ p(X,D)(n)\cdot r^{-n} & \text{otherwise} \end{cases}$$

$$Q(X,D)(n) \hat{=} \begin{cases} (q(X,D)(n)-1)\cdot r^{-n} & \text{if } p(X,D)(n) < 0 \\ (q(X,D)(n)+1)\cdot r^{-n} & \text{if } p(X,D)(n) = D \\ q(X,D)(n)\cdot r^{-n} & \text{otherwise} \end{cases}$$

Expanding the definitions of P and Q, and using lemma 2, we prove:

Theorem 5 (correctness). $\dfrac{X}{D} = Q(X,D)(n) + \dfrac{P(X,D)(n)}{D}$

Similarly expanding the definition of P and using lemma 3, we prove:

Theorem 6 (convergence). $\dfrac{P(X,D)(n)}{D} < r^{-n}$

Thus, $Q(X,D)(n)$ contains n radix-r digits of the quotient $\dfrac{X}{D}$.

3.3 Example instantiations

A number of quotient selection functions have been developed for use with the general subtractive division algorithm. These include three radix-2 algorithms: restoring, nonrestoring, and SRT; and five radix-4 algorithms: two using selection constants for comparison with p (using $a = 3$ and $a = 2$), and three distinct radix-4 SRT lookup tables. The tables developed are for

1) Partial remainder prediction or use of a carry-save adder: approximation of p for error from either computing an estimate of the next remainder as in Taylor's circuit [19] (previously verified in [17]), or error from a carry-save adder [5]
2) Use of a signed-digit adder: approximation of p always overestimates, as would be the case when using signed-digit adders to accumulate the remainder [5]
3) Folded lookup table: lookup table folded for reduced real estate

The process of verification amounts to proving that each quotient digit select function satisfies the type constraints of *qs_type*. The radix-2 instantiations and the radix-4 using selection constants were straightforward verifications. The process of verifying lookup tables in PVS was first illustrated by Rueß, Srivas, and Shankar [17] and repeated here with respect to the general division algorithm. However, the development of two new lookup tables proved insightful in demonstrating the usefulness of PVS in the design process. Not only were omissions from the table detected, but PVS allowed the faster development of new tables. Each table was developed starting from the base table in [17] and then modified to meet the new requirements. PVS was used to indicate which table entries were incorrect under the new specifications, allowing the designer to easily modify the table. Each new table was fully designed and verified in less than three hours elapsed time.

4 Verification with respect to the IEEE standard

This section illustrates how to extend the above general algorithm to provide a verified IEEE compliant implementation of division. This does not constitute a full proof of compliance, it just illustrates how non-exceptional cases of division can be realized. This verification step is performed with respect to a formal specification of IEEE 854 defined using PVS [14]. The verification is performed in two stages. First, a generalization of the basic guard, round, and sticky bit rounding algorithm is shown to satisfy the requirements of the standard. Then, the general subtractive algorithm is shown to provide sufficient information to utilize this rounding scheme. The verified rounding scheme is applicable for all floating-point algorithms. In addition, the theory mapping the standard to the general subtractive algorithm includes a number of intermediate results that apply to all floating point division algorithms.

4.1 Rounding scheme

The IEEE Standards for floating-point arithmetic [6, 7] require support for four rounding modes. The default mode is *round to nearest even*, and requires that the return value be the floating point number nearest to the exact result. If the exact result is halfway between two floating point numbers, the standards require that it be rounded to the one with an even least significant digit. The other three modes round the result towards positive infinity, negative infinity, or zero. The discussion in this section uses the following fact about real numbers.

Lemma 7. *For any integer $b > 1$, a nonzero real number z can be uniquely decomposed into three parts: a sign,$sgn(z) \in \{-1, 1\}$; an integer exponent $e(z)$; and a significand $1 \leq sig(z) < b$ such that*

$$z \;=\; sgn(z) \times sig(z) \times b^{e(z)}$$

The following definition is from the PVS prelude:

Definition 8. The fractional part of a real number z is defined by

$$\{z\} \;\hat{=}\; z - \lfloor z \rfloor$$

The next two definitions are from Miner's PVS formalization of IEEE 854 [14]. The function, *round*, converting real number z to an integer for each of the four rounding modes required by the IEEE standard is defined by:

Definition 9.

$$\begin{aligned}
round(z, to_pos) &\;\hat{=}\; \lceil z \rceil \\
round(z, to_neg) &\;\hat{=}\; \lfloor z \rfloor \\
round(z, to_zero) &\;\hat{=}\; sgn(z) \times \lfloor |z| \rfloor \\
round(z, to_near) &\;\hat{=}\; \begin{cases} \lfloor z \rfloor & \text{if } \{z\} < 1 - \{z\} \\ \lceil z \rceil & \text{if } 1 - \{z\} < \{z\} \\ 2 \times \left\lfloor \frac{\lceil z \rceil}{2} \right\rfloor & \text{otherwise} \end{cases}
\end{aligned}$$

This definition is extended to round a real number z to a real number with at most p base-b digits of precision using the following function:

Definition 10.

$$round\_scaled(z, mode) \;\hat{=}\; round(z \cdot b^{p-1-e(z)}, mode) \cdot b^{e(z)-(p-1)}$$

A common algorithm for implementing IEEE compliant rounding to p digits of precision uses two extra digits and a sticky-flag [11]. The first extra digit is called the guard digit and ensures that the computed result can be normalized while preserving p digits of precision. This is necessary for multiplication and division algorithms. The second extra digit is called the round digit, and is used to control rounding for every mode except *to_zero*. Finally, a sticky flag is required to distinguish the case when the infinitely precise result lies halfway between

two representable values for mode *to_near*. The PVS theory implementing the Guard, Round, Sticky (GRS) scheme has been generalized to allow an arbitrary even radix. Thus it works for both base-2 and base-10 instances of IEEE 854. The principle function realizing the GRS rounding scheme is:

Definition 11. For $s \in \{-1, 1\}$, $n \in \mathbf{N}$, base-b digit d, and boolean *sticky*

$$
\begin{aligned}
\mathit{grs}(s, n, d, \mathit{sticky}, \mathit{to\_pos}) &\hat{=} \begin{cases} n+1 & \text{if } s \geq 0 \wedge ((d > 0) \vee \mathit{sticky}) \\ n & \text{otherwise} \end{cases} \\
\mathit{grs}(s, n, d, \mathit{sticky}, \mathit{to\_neg}) &\hat{=} \begin{cases} n+1 & \text{if } s < 0 \wedge ((d > 0) \vee \mathit{sticky}) \\ n & \text{otherwise} \end{cases} \\
\mathit{grs}(s, n, d, \mathit{sticky}, \mathit{to\_zero}) &\hat{=} n \\
\mathit{grs}(s, n, d, \mathit{sticky}, \mathit{to\_near}) &\hat{=} \begin{cases} n+1 & \text{if } (d > \frac{b}{2}) \vee (d = \frac{b}{2} \wedge (\mathit{sticky} \vee \mathrm{odd?}(n))) \\ n & \text{otherwise} \end{cases}
\end{aligned}
$$

To complete the definition of rounding using the GRS scheme, we use the following functions to scale the result and extract the relevant fields.

Definition 12.

$$
\begin{aligned}
\mathit{scaled}(z) &\hat{=} \lfloor |z| \times b^{p-1-e(z)} \rfloor \\
\mathit{round\_digit}(z) &\hat{=} \lfloor |z| \times b^{p-e(z)} \rfloor \bmod b \\
\mathit{sticky\_flag}(z) &\hat{=} \{|z| \times b^{p-e(z)}\} \neq 0
\end{aligned}
$$

Function *scaled* extracts the p most significant base b digits of z scaled to a natural number. Function *round_digit* extracts the $(p+1)$th digit, and *sticky_flag* is true if and only if there are significant digits beyond the $(p+1)$th. For a result in a sign and magnitude representation, functions *scaled* and *round_digit* can be realized in hardware by extracting the appropriate digits from the given result. Implementing the *sticky_flag* requires some additional logic; the actual realization will vary depending upon the algorithm employed to compute the result. For comparison to *round_scaled*, we define

Definition 13.

$$
\begin{aligned}
&\mathit{round\_grs}(z, \mathit{mode}) \\
&\quad \hat{=} \quad \mathit{sgn}(z) \times b^{e(z)-(p-1)} \times \\
&\qquad \mathit{grs}(\mathit{sgn}(z), \mathit{scaled}(z), \mathit{round\_digit}(z), \mathit{sticky\_flag}(z), \mathit{mode})
\end{aligned}
$$

The principle result of this section is:

Theorem 14.

$$\mathit{round\_grs}(z, \mathit{mode}) = \mathit{round\_scaled}(z, \mathit{mode})$$

The PVS proof of this result consists of a fairly simple case analysis, except for mode *to_near*.

The initial PVS proof for mode *to_near* included a complicated case analysis, where it was difficult to exploit symmetry. IEEE floating point numbers are defined using a sign and magnitude representation. However, the definition of *round_scaled* does not take advantage of this representation. Thus, the first proof for mode *to_near* included an unnatural case split on the sign of the argument. Since the GRS rounding scheme is defined in terms of a sign and magnitude representation, the cases for negative arguments do not align in the same manner as for the positive arguments. Thus, there was little opportunity to reuse proofs from the corresponding positive cases. The PVS proof has been simplified using the following lemma (which was proven using PVS strategy (grind))

Lemma 15.

$$round(z, to\_near) = sgn(z) \times round(|z|, to\_near)$$

Even without the case split due to the sign, the proof for mode *to_near* still involves a difficult case analysis. This case analysis consists of relating the values of the round digit and sticky flag to the corresponding cases from the specification for rounding mode *to_near*.

4.2 Relating the general algorithm to the standard

The verified rounding scheme asserts that in order to achieve IEEE compliant rounding to p significant digits, it is sufficient to compute $(p+1)$ truncated significant digits and determine if there are any remaining non-zero digits. The general subtractive division algorithm ensures at least $n-1$ digits of precision after n iterations. Furthermore, if the computed remainder is nonzero, then there are additional digits in the infinitely precise result. Since it is possible for the radix of the division algorithm to be different from that for the representation of floating point numbers, the PVS theory has to relate the potential division radices to those allowed by the standards.

There are some simple results that describe the range of possible values for floating point division. Let x denote a base-b floating point number with p significant digits. A finite floating point number x is represented using three fields: a sign $\sigma_x \in \{0,1\}$, an integer exponent E_x, and a significand d_x, where d_x is a function from $\{0,\ldots,(p-1)\}$ to $\{0,\ldots,(b-1)\}$. The numerical value of d_x is given by

$$v_d(x) = \sum_{i=0}^{p-1} d_x(i) \cdot b^{-i}$$

The numerical value of floating point number x is given by

$$v(x) = (-1)^{\sigma_x} \cdot b^{E_x} \cdot v_d(x)$$

A floating point number x such that $1 \leq v_d(x) < b$ is normalized. Furthermore, if its exponent also falls within the range allowed by the standard, it is a *normal* floating point number.

The IEEE standard requires that all operations be performed as if to infinite precision and then rounded. The infinitely precise quotient of two floating point numbers x and y is $v(x)/v(y)$. A number of general facts about floating point division have been proven using PVS. These include the following:

Lemma 16.

$$\frac{v(x)}{v(y)} = (-1)^{\sigma_x - \sigma_y} \cdot b^{E_x - E_y} \cdot \frac{v_d(x)}{v_d(y)}$$

This lemma states that a floating point division algorithm can be decomposed into simple operations on the sign and exponent fields combined with an algorithm to compute the quotient of the significands.

Lemma 17. *For normalized floating point numbers x and y,*

$$b^{-1} + b^{-(p+1)} \quad \leq \quad \frac{v_d(x)}{v_d(y)} \quad \leq \quad b - b^{(1-p)}$$

Lemma 18. *For normalized floating point numbers x and y, if $v_d(x) < v_d(y)$ then*

$$\frac{v_d(x)}{v_d(y)} \quad \leq \quad 1 - b^{(1-p)}$$

Lemmas 17 and 18 assert that the result of a floating point division algorithm requires at most one left shift to normalize the result, and furthermore, the necessity of post-divide normalization can be determined by comparing the magnitude of the significands. Therefore, rounding cannot affect the exponent field for a division operation. These results are true for any floating point division algorithm; they do not depend on a particular instantiation.

These results allow us to define the following exponent adjustment function for the quotient of normalized floating point numbers x and y:

$$Adj(x, y) = \begin{cases} 1 \text{ if } v_d(x) < v_d(y) \\ 0 \text{ otherwise} \end{cases}$$

The above results allow us to define a template for a floating point division algorithm

Definition 19. For normal floating point numbers x and y such that $E_{\min} \leq E_x - E_y - Adj(x, y) \leq E_{\max}$, let $z = \frac{v_d(x)}{v_d(y)}$ in

$$div\_algorithm(x, y, mode) \hat{=} finite(XOR(\sigma_x, \sigma_y), E_x - E_y - Adj(x, y), digit\_div(x, y)$$

where

$$digit\_div(x, y) \hat{=} nat2d(grs((-1)^{\sigma_x - \sigma_y}, scaled(z), round\_digit(z), sticky\_flag(z), mode)$$

Constructor *finite* generates a finite floating point number from a sign, an exponent, and a digit vector. Function *nat2d* converts a natural number n, where $n < b^p$, to a digit vector. To complete the algorithm, it is necessary to compute *scaled*, *round_digit*, and *sticky_flag* for the given quotient.

The theory relating the general algorithm to the IEEE standard uses the same formal parameters as the the general division theory. It also uses additional parameters needed to import the IEEE theories. The base for the floating point representation is b, the radix for the division algorithm, r, is selected so that $r = b^i$ for some positive integer i. This ensures shift operations can be effectively used in the computation of the partial remainder. The divisor, $v_d(y)$, already satisfies the type constraint *D_type*. The dividend needs to be shifted right either one or two base-b digits to ensure that it satisfies the initial type constraints for the partial remainder. Finally, the algorithm must be be iterated enough times to ensure $p + 1$ significant digits. The following definitions invoke the general algorithm using arguments selected to ensure the algorithm computes enough information for IEEE compliant rounding.

Definition 20. Let $pre = 2 - \lfloor \rho \rfloor$, $i = \log_b(r)$, and $steps = \lceil \frac{p+2+pre}{i} \rceil$ in

$$\begin{aligned} \text{Quotient}(x, y) &\hat{=} b^{pre} \cdot Q(v_d(x) \cdot b^{-pre}, v_d(y))(steps) \\ \text{Remainder}(x, y) &\hat{=} b^{pre} \cdot P(v_d(x) \cdot b^{-pre}, v_d(y))(steps) \end{aligned}$$

Using results from the general division theory, we establish the following results

Lemma 21.

$$\frac{v_d(x)}{v_d(y)} = Quotient(x, y) + \frac{Remainder(x, y)}{v_d(y)}$$

Lemma 22.

$$\frac{Remainder(x, y)}{v_d(y)} < b^{-(p+2)}$$

This result combined with lemma 17 ensures that the computed quotient has a sufficient number of significant digits for correct rounding. The arguments for the GRS rounding algorithm are:

Definition 23.

$$\begin{aligned} scaled\_div(x, y) &= \lfloor \text{Quotient}(x, y) \cdot b^{(p-1)+Adj(x,y)} \rfloor \\ round\_digit\_div(x, y) &= \lfloor \text{Quotient}(x, y) \cdot b^{p+Adj(x,y)} \rfloor \bmod b \\ sticky\_flag\_div(x, y) &= (\{\text{Quotient}(x, y) \cdot b^{p+Adj(x,y)}\} + \text{Remainder}(x, y)) > 0 \end{aligned}$$

All that remains is to prove the following:

Theorem 24 (IEEE_compliant). *For normal floating point numbers x and y such that $E_{\min} \leq E_x - E_y - Adj(x, y) \leq E_{\max}$*

$$fp\_div(x, y, mode) = div\_algorithm(x, y, mode)$$

The PVS proof of IEEE_compliant has two major parts. The first part of the proof consists of showing that the restrictions on the exponents of x and y ensure that the result of *fp_div* returns a normal floating point number. The second part of the proof uses the following identities:

Lemma 25.

$$scaled\_div(x, y) = scaled\left(\frac{v(x)}{v(y)}\right)$$

$$round\_digit\_div(x, y) = round\_digit\left(\frac{v(x)}{v(y)}\right)$$

$$sticky\_flag\_div(x, y) = sticky\_flag\left(\frac{v(x)}{v(y)}\right)$$

The PVS proofs of these three identities involve algebraic manipulation of experessions composed of exponents, absolute values, and the integer floor function. The proofs are not conceptually difficult, but they do not succumb to PVS' brute force proof strategies.

4.3 IEEE compliant division

Given the above verified general algorithm and a quotient selection function of type *qs_type* for some division radix r, quotient digit set bound by a, and floating point base b. it is trivial to construct a verified IEEE compliant subtractive division algorithm.

An example instantiation for IEEE 754 single precision radix-4 SRT division using signed-digit adders to compute the partial remainder consists of the following sequence of PVS declarations.

```
IMPORTING
  divide_types[4,2,2],
  signed_digit_lookup

qs: qs_type[4,2,2] = q_sel

IMPORTING
  ieee_divide[4,2,2,qs,24,192,127,-126]
```

Theory `signed_digit_lookup` contains the definition of an SRT4 lookup table suitable for use with a signed-digit adder. In addition, it contains the proof that the quotient selection function *q_sel* satisfies type *qs_type[4,2,2]*. Theory `ieee_divide` defines function *div_algorithm* and includes the proofs from the previous section. With the instantiation of the quotient selection function, the function *div_algorithm* completely describes a verified IEEE compliant division algorithm composed of operations that have simple hardware realizations. The designer must verify that the hardware design correctly implements this algorithm, but need not be concerned with the functional correctness of the algorithm.

Given a collection of verified quotient selection functions, it is easy for a designer to select one appropriate to the constraints of a particular development effort.

5 Suggestions for future work and lessons learned

5.1 Suggestions for future work

There are a number of ways to build on the work presented here. The basic style of the specification is common for a large number of floating point algorithms. The most obvious class of algorithms to consider is subtractive square root algorithms. Another good candidate for exploration is whether a similar general schema can be developed for division through multiplication algorithms.

Another example to be considered is the generalized CORDIC algorithm [22]. The algorithm has already been defined in PVS, and the solutions to the defining CORDIC equations have been verified. These proofs required the addition of some axioms describing properties of trigonometric and hyperbolic functions. The limited support (in PVS) for reasoning about these functions made analysis of accumulated error and convergence difficult, thus remaining as future work.

From the above mentioned work, it should be possible to build up a library of general floating point algorithms verified with respect to the IEEE standard. Such a library would present a developer of floating point hardware with a variety of options, provided there was a good link between the verified library and hardware development tools.

5.2 Lessons learned

In general, the verification of functional correctness of recurrence based algorithms is a simple matter using PVS. However, details often ignored by other verifications required the most difficult and time consuming proofs. There were two primary sources of difficulty: poor choices in the structure of the formal specification and working in a domain where there is limited support from the prover. During this verification effort, most proof steps consisted of algebraic manipulation of expressions involving exponentiation for which there are a limited number of preproven properties. Although the prelude includes some facts about exponentiation, there are a number of improvements possible to more effectively automate such proofs.

The PVS type system can be used to effectively restrict types in definitions. However, this may lead to much time spent proving irrelevant type correctness conditions. Finding the right balance can be difficult. During the verification relating the algorithm to the IEEE standard, much of the effort consisted of repeatedly discharging the same collection of type correctness conditions.

6 Concluding remarks

Formal verification is an enabling technology. This general verification will allow designers to focus their efforts on more advanced optimizations of hardware implementations secure in the knowledge that the routine aspects of the design have been addressed. However, a great deal of work is still needed to make formal verification a useful technology. In particular, a designer should not be required to generate all of the supporting theories for well known algorithms and hardware structures. A large set of libraries should be available from which to select the pieces required to complete and verify a design.

The ultimate goal of the work is to assist in the development of verified hardware. With that in mind, the general algorithm was presented in a standard form that can be easily transformed to an equivalent tail-recursive description [10]. This would provide a top-level specification for deriving a hardware realization using a transformational system such as DRS [3]. This process has been tested with the general subtractive division algorithms presented in this paper.

The work presented here is a major step toward establishing an environment conducive to development of provably correct floating point hardware. The primary benefit of the work is not the fact that subtractive division algorithms are shown to be compliant with IEEE standards. Instead, this work demonstrates that with proper foresight in developing and verifying generalized solutions, it is then much easier for future designers to verify particular instantiations of those general solutions. As a complete library of verified floating point algorithms emerges, future floating point implementations should have a much higher confidence of correctness.

Acknowledgements

We are grateful for the helpful comments and suggestions made by Steve Johnson, Mandayam Srivas, and the anonymous reviewers.

References

1. Geoff Barrett. A formal approach to rounding. In *Proceedings 8th Symposium on Computer Arithmetic*, pages 247–254, 1987.
2. Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.
3. Bhaskar Bose. DRS - derivational reasoning system: A digital design derivation system for hardware synthesis. In T. Hilburn, G.Suski, and J. Zalewski, editors, *Safety and Reliability in Emerging Control Technologies*. Elsevier Science Publishers, Oxford, UK, 1996. ISBN 0 08 042610 7, to be published.
4. E. M. Clarke, S. M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag. To appear.

5. Miloš D. Ercegovac and Tomás Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations.* Kluwer Academic Publishers, 1994.
6. IEEE-754. *Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Std 754-1985.
7. IEEE-854. *Standard for Radix-Independent Floating-Point Arithmetic*, 1987. ANSI/IEEE Std 854-1987.
8. J. Harrison. Floating point verification in HOL. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Proceedings 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 186–199, Aspen Grove, UT, USA, September 1995. Springer Verlag.
9. J. O'Leary, M. Leeser, J. Hickey, and M. Aagaard. Non-restoring integer square root: A case study in design by principled optimization. In T. Kropf and R. Kumar, editors, *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 of *Lecture Notes in Computer Science*, pages 52–71, Bad Herrenalb, Germany, September 1994. Springer Verlag. published 1995.
10. Steven D. Johnson. *Synthesis of Digital Design from Recursion Equations.* The MIT Press, Cambridge, MA, 1984. ACM Distinguished Dissertation 1984.
11. Israel Koren. *Computer Arithmetic Algorithms.* Prentice Hall, 1993.
12. M. Leeser and J. O'Leary. Verification of a subtractive radix-2 square root algorithm and implementation. In *Proceedings International Conference on Computer Design 1995 (ICCD '95)*, pages 526–531, October 1995.
13. Irwin Meisels and Mark Saaltink. The Z/EVES reference manual (draft). Technical Report TR-95-5493-03, ORA Canada, December 1995.
14. Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA, Langley Research Center, Hampton, VA, June 1995.
15. J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5$_k$86 floating point division algorithm. (Submitted), URL:http://devil.ece.utexas.edu:80/~lynch/divide/-divide.html, March 1996.
16. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
17. H. Rueß, N. Shankar, and M.K. Srivas. Modular verification of SRT division. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag. To appear.
18. David M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating-point square root algorithm. unpublished manuscript, July 1996.
19. George S. Taylor. Compatible hardware for division and square root. In *Proceedings 5th Symposium on Computer Arithmetic*, pages 127–134, 1981.
20. V. Pratt. Anatomy of the pentium bug. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107, Aarhus, Denmark, May 1995. Springer Verlag.
21. D. Verkest, L. Claesen, and H. De Man. A proof of the nonrestoring division algorithm and its implementation on an ALU. *Formal Methods in System Design*, 4:5–31, January 1994.
22. J. S. Walther. A unified algorithm for elementary functions. In *Proceedings of Spring Joint Computer Conference*, pages 379–385, 1971.

This article was processed using the LaTeX macro package with LLNCS style