, , 1–28 ()

Modular Verification of SRT Division *

HARALD RUESS ruess@informatik.uni-ulm.de
Universität Ulm, Fakultät für Informatik, 89069 Ulm, Germany

NATARAJAN SHANKAR AND MANDAYAM K. SRIVAS {shankar, srivas}@csl.sri.com
SRI International Computer Science Laboratory, Menlo Park, CA 94025 USA

Abstract. We describe a formal specification and mechanized verification in PVS of the general theory of SRT division along with a specific hardware realization of the algorithm. The specification demonstrates how attributes of the PVS language (in particular, predicate subtypes) allow the general theory to be developed in a readable manner that is similar to textbook presentations, while the PVS `table` construct allows direct specification of the implementation's quotient lookup table. Verification of the derivations in the SRT theory and for the data path and lookup table of the implementation are highly automated and performed for arbitrary, but finite precision; in addition, the theory is verified for general radix, while the implementation is specialized to radix 4. The effectiveness of the automation stems from the tight integration in PVS of rewriting with decision procedures for equality, linear arithmetic over integers and rationals, and propositional logic. This example demonstrates that the resources of an expressive specification language and of a general-purpose theorem prover are not inimical to highly automated verification in this domain, and can contribute to clarity, generality, and reuse.

Keywords: Computer Arithmetic, Hardware Verification, SRT Division

1. Introduction

The SRT division algorithm is one of the most popular methods for implementing floating-point division and related operations in high-performance arithmetic units. Even though the theory of SRT division has been extensively studied [2], the design of dividers still remains a serious challenge [23], and it is easy to make mistakes in its implementation—as was highlighted by the much publicized FDIV error in the Intel Pentium chip. Pratt [27] points in his analysis that it is unlikely testing alone would have caught that error as it was due to five wrong entries in the quotient lookup table in a region of the table that was thought to be unreachable. In this paper, we demonstrate that machine-assisted formal verification can be effectively used in the design and debugging of arithmetic circuits.

We present a formal development and mechanized verification of a general SRT division algorithm and an implementation of it based on the circuit given by Taylor [34]. This circuit implements the IEEE floating-point standard, and its kernel consists of a fixed-point iteration. The verification of this kernel was performed in the interactive theorem proving system PVS [25]. Since our goal is to perform the

* Supported in part by DARPA under Darpa Order A721 through NASA Ames Research Center under contract NASA-NAG-2-891, and by NSF Grants CCR-9300444 and CCR-9712383. A shorter version of this paper appeared in the proceedings of CAV'96 [29]. This work was done while the first author was visiting the SRI International Computer Science Laboratory.

verification so that most of the initial set-up effort can be reused in the verification of other similar circuits, we take a modular approach that separates concerns about general facts of the SRT theory from a specific circuit implementation and a lookup table. Furthermore, we relate the different levels of abstraction by deriving constraints that must be satisfied by the lower levels so that each level can be verified separately.

More precisely, the formalization and verification of the SRT divider proceeds in two steps. First, we formalize textbook knowledge about SRT dividers at an algorithmic level and verify its correctness. The formalization at this level does not use a specific data path to compute the partial remainder nor a specific lookup table. It characterizes a set of semantic constraints a lookup table for the quotient digit ought to satisfy and the recurrence relation that a partial remainder computation circuit ought to preserve. These constraints on quotient digit selection in terms of an approximate partial remainder and divisor are systematically derived from the correctness and convergence requirements. This part of the verification is at a level of abstraction where all the variables range over integers and rational numbers and where the radix and redundancy factor for the quotient digit have not been fixed.

In the second step, we specify a data path circuit based on the implementation given by Taylor [34]. This circuit is defined in terms of bit-vector signals over time and computes the partial remainder, an approximate partial remainder, and the quotient digit using a specific quotient digit lookup table. We then show that the data path circuit and the lookup table meet the constraints characterized in step one. Both steps of the verification are performed for an arbitrary but finite precision which appears as a parameter to the specification. The first step of the verification is applicable to arbitrary radixes, while the second step assumes a radix-4 implementation, since it uses a lookup table for radix-4. Note that the arithmetic units in the datapath are specified in terms of arithmetic constraints relating the arithmetic interpretation of their input and output signals. The actual gate-level hardware implementations of the datapath and lookup table are not considered here.

In developing the theory of SRT division and deriving a specific implementation corresponding to this theory, we have made significant use of the language features of PVS and its highly integrated automated deductive capabilities. The language features we have exploited include the higher-order logic, predicate subtypes and dependent types, and a construct for defining lookup tables. The deductive capabilities include a simplifier which integrates the use of arithmetic and equality decision procedures with term rewriting. Our main conclusion is that the theory and implementation details of SRT division can be elegantly captured in an expressive specification language and formally verified with a high degree of automation.

We first survey some of the related work on the mechanized verification of computer arithmetic, particularly division, circuits in Section 2. PVS is briefly introduced in Section 3 and the theory underlying SRT division is presented in Section 4. The PVS formalization of this theory along with the lemmas and proofs is described in Section 5. The second part of the verification involving the datapath and lookup table is presented in Sections 6 and 7. Section 8 contains some concluding observations. Some of the details of the PVS formalization are relegated to the Appendices.

2. Related Work

Verkest, Claesen, and de Man [36] and Leeser and O'Leary [19] have used theorem provers to verify a non-restoring divider and a radix-2 subtractive square root algorithm, respectively. The circuits verified in both of these efforts are not based on the SRT method and hence do not contain the kinds of optimizations used in SRT division. German and Clarke [8,13] performed a verification of Taylor's SRT division circuit considered in this paper by deriving a set of inequalities that the circuit imposes on the data path signals and then showing, in the MAPLE symbolic algebraic system, that two main SRT correctness invariants are preserved by the data path inequalities. This work provided the main impetus for our work (see also [29]). Clarke, German, and Zhao [9] have subsequently and independently mechanized their verification of Taylor's circuit using the ANALYTICA theorem prover augmented with an implementation of the SUP-INF method [4,33] for linear arithmetic. Our work goes beyond a verification of a specific SRT circuit by also formalizing the general theory of SRT division within a modular framework that can be used to verify other similar circuits. Though we use the same datapath assumptions as Clarke, German, and Zhao, we describe the hardware implementation more concretely in terms of bit-level datapath signals and a complete lookup table. The verification of the invariants by Clarke, German, and Zhao is fully automatic but on a restricted set of theorems, whereas our analysis is more general but requires some manual guidance. In our proofs within the abstract SRT theory, manual guidance is needed to invoke specific lemmas and to cope with nonlinear arithmetic. In the concrete proofs, manual guidance is needed to circumvent some inefficiencies in the PVS decision procedures in dealing with bit-level representations.

The above verification efforts [9,29] are restricted to the fixed-point kernel of the division circuit. Recently, Miner and Leathrum [21] used PVS to verify a general class of subtractive division algorithms as well as specific instances in this class including Taylor's SRT circuit, with respect to the IEEE floating-point arithmetic standards [15,16]. They also showed how PVS could be used in the rapid design of new division circuits starting from existing ones.

Methods based on ordered BDDs and symbolic model checking are not well-suited for verifying multipliers and dividers since BDD graphs for such operations grow exponentially with the word size [3]. However, Bryant [5] has used BDDs to check the relation that one iteration of the SRT circuit must preserve for the circuit to divide correctly. To do the verification, he needed to construct a gate-level representation of a *checker-circuit* (much larger than the verified circuit) to describe the desired behavior of the verified circuit, which is not the ideal level of specification.

Bryant and Chen showed that a data-structure similar to BDDs called binary moment diagrams (BMDs) could be used to efficiently verify multipliers against their number-theoretic specification [3]. Since BMDs can be inefficient for checking inequalities over bit-vectors, Clarke, Fujita, and Zhao use an extension of BDDs called *hybrid decision diagrams* [7] to represent integer functions and check relations on them. Clarke, Khaira, and Zhao [11] have used hybrid decision diagrams to extend the symbolic model-checking algorithm used in SMV to express and verify word-level properties on numbers (see also [6, 18] for practical applications of word-level model checking). Their word-level model-checker can be used to check if finite-sized arithmetic circuits satisfy the desired number-theoretic properties. It has been used to verify Taylor's SRT circuit by checking that a state transition model of the circuit satisfies the main SRT invariants. Word-level model checking [11] can be applied only to datapaths of a specific size, whereas theorem proving can be parametric in the width of the datapath.

There is a large body of loosely related work on the deductive verification of computer arithmetic circuits that does not deal with SRT division. Moore, Lynch, and Kaufmann [22] verified the floating point division microcode consisting of 32 microinstructions for the $\mathrm{AMD5}_K86$ microprocessor using ACL2.[1] Subsequently, Russinoff [31] verified the square root microcode for the same processor. There are several papers describing the verification of multiplier circuits using various theorem provers [10, 17, 30].

3. An Overview of PVS

The PVS system combines an expressive specification language with an interactive proof checker based on automated procedures for simplification and rewriting using decision procedures. It has been used for reasoning in domains as diverse as microprocessor verification, protocol verification, and algorithms and architectures concerning fault-tolerance [25].

The PVS specification language builds on classical typed higher-order logic with the usual base types `bool`, `nat`, `rational`, `real`, ... and the function type constructor `[A -> B]`. The type system of PVS is augmented with *dependent types* and *abstract data types*. A built-in *prelude* and loadable *libraries* provide a base of specifications and proven facts for a large number of theories. PVS specifications are packaged as *theories* that can be parametric in types and constants. The parameterized theory `fp2c` in $\boxed{1}$, for example, defines an interpretation `val` of bit-vectors as 2's-complement binary fixed-point numbers with `k` integral and `m` residual bits.

```
fp2c[k: posnat, m: nat]: THEORY                                          1

BEGIN

  IMPORTING bitvectors@bv, bitvectors@bv_int

  val(bv: bvec[k + m]): rational =
    2^(-m) * (-2^(k + m - 1) * bv(k + m - 1) + bv2nat_rec(k + m - 1, bv))

END fp2c
```

This theory imports the bit-vector theories `bv` and `bv_int` from the bit-vector library `bitvectors`. Bits are represented as elements of type `upto(1)` which consists of the numbers `0` and `1`, and a bit-vector of length `N` is represented by finite functions of type `[below(N) -> bit]`. The definition of `val` uses the unsigned interpretation `bv2nat_rec` of bit-vectors from the bit-vector library. The function `val` assigns a numeric interpretation to a `k + m` bit representation `bv` by

1. Interpreting the leading sign bit `bv(k + m - 1)` as -2^{k+m-1} as intended of a 2's-complement integer representation.

2. Taking the unsigned integer interpretation of the remaining bits

$$\mathtt{bv}(0) + \mathtt{bv}(1) * 2 + \mathtt{bv}(2) * 2^2 \ldots \mathtt{bv}(k + m - 2) * 2^{k+m-2}.$$

3. Dividing the 2's-complement $k+m$ bit interpretation given by the sum of steps 1 and 2 by 2^m (i.e., multiplying by 2^{-m}) to obtain the value for an interpretation with k integral bits and m residual bits.

Predicate subtypes of the form {`x:A | P(x)`} are a significant part of the type system of PVS. These subtypes consist of exactly those elements `a` of type `A` satisfying predicate `P(a)`. Predicate subtypes are used to explicitly constrain the domain and ranges of operations in a specification and to define partial functions. Predicate subtypes are used to define the integer ranges `upto(n)` ($[0, \ldots, n]$), `subrange(-m, n)` ($[-m, \ldots, n]$), and the strictly positive rationals `posrat` used in this specification. In general, type-checking with predicate subtypes is undecidable, and the type-checker generates *type correctness conditions* (TCCs) corresponding to predicate subtypes. A large number of TCCs are discharged by specialized proof strategies, and a PVS expression is not considered to be fully type-checked unless all generated TCCs have been proven correct. In the theory `fp2c` above, there is a generated TCC to ensure that `k + m - 1` is in the subrange `0` to `k + m`. This and many other similar TCC proof obligations are discharged automatically by PVS, and any remaining TCCs can be proved with manual guidance.

Proofs in PVS are presented in a sequent calculus. The atomic commands of the PVS prover component include induction, quantifier instantiation, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. The `skolem*` command, for example, repeatedly introduces Skolem constants for universal-strength quantifiers, and `assert` combines rewriting with decision procedures.

Finally, PVS has an LCF-like [14] strategy language for combining inference steps into more complicated proof strategies. The strategy `then`, for example, applies a sequence of sub-strategies, `case*` case splits for a given list of formulas along all the different branches, and the defined rule `grind` combines rewriting with propositional simplification using BDDs and decision procedures [12].

4. SRT Division

Digit recurrence algorithms use subtraction as the iterative operator. The quotient is represented in radix-r form and one quotient digit is calculated in each iteration. This class can be further divided into *restoring* and *nonrestoring* division. Restoring division is similar to the familiar paper-and-pencil division where a non-negative partial remainder is repeatedly shifted and reduced by the product of the quotient digit and the divisor until the remainder goes negative in which case it has to be restored. Nonrestoring division eliminates the restoration cycles by, for example, allowing negative partial remainders. This can be accomplished by allowing positive as well as negative values of the quotient digits so that an overestimation of the quotient digit in one iteration can be corrected in subsequent iterations through the generation of negative quotient digits. The SRT class of dividers [20, 28, 35] constitute a widely used technique for implementing efficient nonrestoring division in high-speed floating point units.

This section presents some fundamental concepts of SRT division. These formalizations are applied in Sections 6 and 7 to prove the correctness of a specific circuit. Note that in this paper we restrict ourselves to the fixed-point division kernel, and ignore, for the time being, typical pre- and postcomputations like scaling, quotient conversion, possibly *on-the-fly*, or rounding.

Subtractive Algorithms. Kernels of IEEE compliant floating-point division circuits typically consist of a fixed-point division of the dividend mantissa p by the divisor mantissa d. In what follows we will assume that both p and d are normalized fractions of the form $1.xx\ldots xx_2$; in particular $1 \leq p, d < 2$. We will assume that the quotient is represented in the base r although both p and d are represented in binary, 2's-complement form. For efficiency reasons, SRT dividers use a *redundant digit set* in the subrange from $-a$ to a for quotient digits q rather than the usual subrange from 0 to $r-1$. The symbol ρ represents the fraction $a/(r-1)$. Typical choices for a and r are 2 and 4, respectively. This way, only the quotient digits $-2, -1, 0, 1, 2$ will be used to multiply the divisor and multiplication by each of these digits is easily implemented in hardware, whereas a representation involving $0, 1, 2, 3$ as quotient digits would entail multiplication by 3 which is not easy.

In each iteration, the partial remainder p_{i+1} is computed from the previous partial remainder and quotient digit using the following recurrence:

$$\begin{aligned} p_0 &= p \\ p_{i+1} &= r * (p_i - q_i * d) \text{ with the restriction } |p_{i+1}/d| \leq r * \rho. \end{aligned}$$

As can be noted from the recurrence above, each iteration of the iteration comprises the following steps:

- determine the quotient digit q_i, and
- compute the next partial remainder p_{i+1}.

The restriction on the recurrence constrains the choice of the quotient digit q_i. We already noted that the quotient digits are taken from the subrange $-a$ to a. This

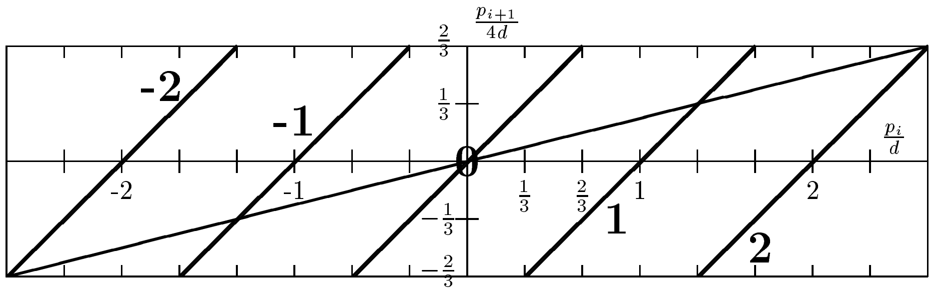

Figure 1. The Robertson Diagram

representation has some redundancy in that there is some leeway in the selection of a quotient digit so that it can be determined from an approximation of the partial remainder p_i.

Example. We consider a simple example of dividing $p = 143/128 = 1.0001111$ by $d = 11/8 = 1.011$. The numbers are represented in binary 2's-complement form with one sign bit and three bits preceding the binary point. The expected answer is $p/d = 13/16 = 0.31_4$.

p_0	=	0 0 0 1	.0 0 0 1 1 1 1	
subtract d	=	1	.0 1 1	set $q_0 := 1$
yields $p_1/4$	=	1 1 1 1	.1 0 1 1 1 1 1	
shift p_1	=	1 1 1 0	.1 1 1 1 1	
add d	=	1	.0 1 1	set $q_1 := \bar{1}$
$p_2/4$	=	0 0 0 0	.0 1 0 1 1	
shift p_2	=	0 0 0 1	.0 1 1	
subtract d	=	1	.0 1 1	set $q_2 := 1$
$p_3/4$	=	0 0 0 0	.0	

The first quotient digit q_0 is generated as 1 even though this is an over-estimation thus leading to a negative value for the next partial remainder p_1. Note that p_1 is obtained by left-shifting the difference of p_0 and $q_0 * d$ by two bits. The over-estimation in q_0 is corrected in the next iteration by generating a negative quotient bit $\bar{1}$. This yields a positive partial remainder which exactly matches the divisor. Consequently, the quotient is calculated as $q = (1.\bar{1}1)_4 = 1-1/4+1/16 = 13/16$.

Convergence. The example shown above does not explain the constraints on quotient selection and how these constraints together with the recurrence guarantee that the algorithm converges to an accurate quotient value after a number of iterations. This can be explained in terms of a Robertson diagram. Both the explanation and the diagram are adapted from Pratt [27].

Fixing a as 2 and r as 4, the Robertson diagram consists of the x-axis ranging from $-8/3$ to $8/3$ (more generally, $-a * r/(r-1)$ to $a * r/(r-1)$), and a y-axis ranging from $-2/3$ to $2/3$ (more generally, $-a/(r-1)$ to $a/(r-1)$). The slope of the main diagonal is $1/4$ (generally, $1/r$) so that a transposition from the y-axis to the x-axis about this diagonal corresponds to a multiplication by 4 (r). The dark q-digit lines have a slope of 1 and are labelled by the quotient digits generated by them. Each i'th cycle of the algorithm begins with the point on the x-axis corresponding to the exact value p_i/d. The quotient digit q_i corresponding to this can be obtained by moving upwards or downwards from the point p_i/d till a q-digit line is intersected. The label for the line yields the digit q_i and the corresponding y-axis value yields the new but unshifted partial remainder $p_{i+1}/4d$. This is easy to see because the slope of a q-digit line is 1 and it intersects the x-axis at q_i so that the y-axis value of the intersection is exactly $p_i/d - q_i$. This partial remainder when transposed about the main diagonal yields the value of p_{i+1}/d for the next iteration.

Several observations about SRT division are directly evident from the Robertson Diagram:

1. The $8/3$ bound on the partial remainder on the x-axis (and hence the $2/3$ bound on the unshifted partial remainder on the y-axis) are tight bounds needed for convergence. If the partial remainder value lies beyond this bound, then the only choices for the quotient digit are -2 on the negative side and 2 on the positive side, and in either case, the partial remainder diverges, i.e., grows beyond any given bound.

2. Conversely, the bound of $8/3$ on the partial remainder together with the division by 4 on each iteration implies that after k iterations, the difference between the accurate quotient p/d and the computed quotient q is bounded by $8/(3 * 4^k)$.

3. For some values of p_i/d, there is some redundancy in the choice of the quotient digit q_i. For example, if p_i/d is between $1/3$ and $2/3$, then the quotient digit q_i can be either 0 or 1. The significance of this redundancy is that it allows the quotient digit q_i to be computed from an approximation of p_i which means that the computation of q_i can be done in parallel with the computation of p_i.

Section 5 formalizes the general theory of SRT division in PVS to prove convergence and to extract the bounds on quotient selection. This theory is then instantiated for specific values for the radix r and the redundancy factor a to derive a specific datapath for SRT division in Section 6. Section 7 describes the verification of a specific quotient lookup table for the division circuit.

5. SRT Division in PVS

In this section, we describe our formalization of the SRT division algorithm in PVS, formally prove the convergence of this algorithm to the correct value, and develop conditions for lookup tables and SRT division circuits that are sufficient to establish the overall correctness of a specific implementation.[2]

These developments are packaged in the PVS theory `srt` in $\boxed{2}$, which is parameterized with respect to meaningful choices of radix `r` and the delimiter `a` of the quotient digit set. The remaining theory parameters `delta` and `eps` are described below. To obtain fast algorithms one uses a *redundant digit set* `subrange(-a, a)` for quotient digits `q`. A common choice for the radix and the delimiter of the quotient set is `r = 4`, `a = 2`, and consequently the redundancy factor `rho` is `2/3`. Notice that the circuit in Section 6 also uses the choices `r = 4` and `a = 2`. PVS comments are indicated by a preceding `%` symbol.

```
srt[r: {r: posnat | r >= 2},                                        2
    a: {a: posnat | r/2 <= a & a < r},
    delta, eps: posrat]: THEORY
BEGIN
 % -- Recurrence

  p, p_new: VAR rational
  d       : VAR posrat
  q       : VAR subrange(-a, a)

  recurrence?(p_new, p, q, d):bool =
    p_new = r * (p - q * d)

  rho: {q: posrat | 1/2 < q & q <= 1} =  % Redundancy Factor
    a / (r - 1)

  p_over_d_bound?(d, p_new): bool =
    -r * rho <= p_new / d & p_new / d <= r * rho

  ...
```

Given a radix `r`, a rational dividend `p` and a rational divisor `d`, one iteration of a radix-`r` SRT algorithm generates a quotient digit `q` and a new partial remainder `p_new` such that the relation `recurrence?(p_new, p, q, d)` in $\boxed{2}$ holds. The choice of the quotient digit `q` is henceforth restricted to values for which `p_over_d_bound?(d, p_new)` holds in order to guarantee convergence of the algorithm. The PVS versions of both the recurrence and the restriction are exactly as shown on page 6.

Convergence. The function `val(i + 1, qq)` in $\boxed{3}$ computes the radix-`r` fixed-point value of the accumulated quotient `qq(0).qq(1)`...`qq(i)` with one leading digit and `i` residual digits. The theorem `convergence` states that this quotient value is an approximation to the infinite precision fraction `(pp(0) / d)` within an error bound.

```
% -- Convergence                                                    3

  qq: VAR sequence[subrange(-a, a)]
  pp: VAR sequence[rational]

  i, j, k: VAR nat

  val(i, qq): RECURSIVE rational =
    IF i = 0 THEN 0
    ELSE  qq(i - 1) * 1/r^(i - 1) + val(i - 1, qq)
    ENDIF
  MEASURE i

  lemma1: LEMMA
    (FORALL j: recurrence?(pp(j + 1), pp(j), qq(j), d))
      IMPLIES pp(0) / d - val(i, qq) = 1/r^i * (pp(i) / d)

  convergence: THEOREM
    ((FORALL j: recurrence?(pp(j + 1), pp(j), qq(j), d))
    AND (FORALL k: p_over_d_bound?(d, pp(k))))
     IMPLIES
       LET residue = pp(0) / d - val(i + 1, qq) IN
        -1/r^i * rho <= residue & residue <= 1/r^i * rho
```

The convergence theorem is an immediate consequence of the invariant `lemma1` and the bound on `(pp(i) / d)` given by `p_over_d_bound?`. The statement of `lemma1` asserts that the residue, namely the difference of the infinite precision quotient `(pp(0) / d)` and the quotient value computed after i iterations is the resulting partial remainder `pp(i)` divided by `d` and scaled by `(1 / r^i)` where `^` is the exponentiation operator defined in the PVS prelude library. The proof of `lemma1` is completed by the single PVS command

```
(induct-and-simplify "i" :THEORIES "real_props")
```

that uses induction on the number of iterations `i` and some basic facts from the library about real and rational numbers.

Quotient Selection. The hard part in each iteration is to determine a *legitimate* quotient digit `qq(i)` so that the next partial remainder `pp(i + 1)` also satisfies the boundary constraint `p_over_d_bound?`. As seen in the Robinson diagram, the quotient `(p / d)` should be within `rho` of the chosen quotient digit `q` so that `p_new / d` is bounded by `r * rho`. This constraint on quotient digit selection is stated in the definition `legitimate?`. The significance of the equivalence is that if the quotient digit is selected according to the `legitimate?` constraint, then the `p_over_d_bound?` constraint that is needed for convergence, holds of the new partial remainder `p_new` computed by the recurrence.

```
% -- Legitimacy                                                    4

 legitimate?(q, d, p): bool =
     q - rho <= p / d  &  p / d <= q + rho

 lemma2: LEMMA
   recurrence?(p_new, p, q, d) IMPLIES
     (p_over_d_bound?(d, p_new) IFF legitimate?(q, d, p))
```

The equivalence transformations needed to establish `lemma2` in $\boxed{4}$ are

```
        q - rho <= p / d <= q + rho
<=>       {subtracting q, multiplying by r}
        -r * rho <= r * (p / d - q) <= r * rho
<=>       { p_new = r * (p - q * d) }
        -r * rho <= p_new / d <= r * rho.
```

For specific interpretations, say `r = 4` and `a = 2`, the combination of decision procedures with rewriting on known facts about real and rational numbers

```
(grind :theories "real_props")
```

discharges the proof obligation `lemma2` in $\boxed{4}$ automatically. In the general case, however, where `r` and `a` are arbitrary parameters, the proof of this fact involves solutions of nonlinear inequalities, and the PVS prover consequently needs some guidance.

```
(then (skosimp*) (expand "legitimate?")
      (both-sides "-" "q!1") (assert)
      (both-sides "*" "r") (assert)
      (grind))
```

Universal quantifier elimination `skosimp*` in the proof script above introduces the constant `q!1`, `expand` unfolds the `legitimate?` definition, and the `both-sides` strategies manipulate the resulting chain of inequalities by subtracting `q!1` from each side followed by multiplications of each side by `r` as in the first two steps of the informal proof above. Finally, the `grind` command fills in the required proof details.

Notice that the boundaries of the selection intervals specified by `legitimate?` depend on the divisor `d`. Figure 2, for example, graphically displays the region for legitimately selecting `q = 1` for the case where `r = 4` and `a = 2` (thus `rho = 2/3`): it is bound by the dashed lines `5/3 * d` and `1/3 * d`.

Redundancy. Shaded regions in Figure 2 indicate pairs (`d`, `p`) for which selection intervals for the quotient digit `q` overlap. This redundancy permits the calculation of `q` from truncated versions `P` and `D` (of type `rational`) of the partial remainder and divisor, respectively.

Note that if (both positive and negative) numbers are represented in 2's complement form, which is what we assume in the circuit we verify later, truncation (after

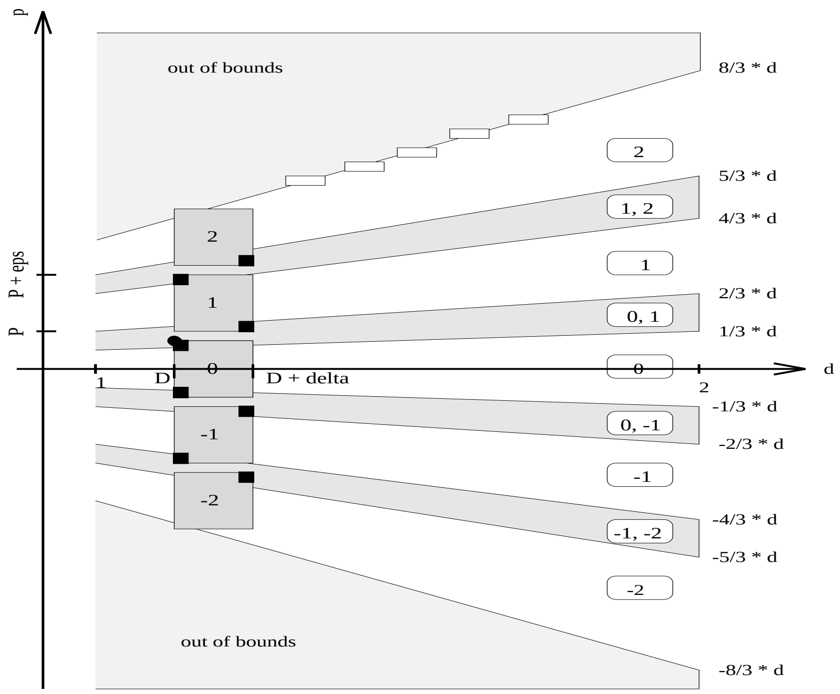

Figure 2. pd-plot for `r = 4` and `a = 2`

the binary point) always yields a number of lesser value. Let `delta` and `eps` be two arbitrary positive rational numbers (see also $\boxed{2}$) such that `P` underestimates p by at most `eps`, and similarly, `d` is at most `D + delta`. Then it is easy to show that `p_over_d_bound?` on `d` and `p` entails the constraint `P_bound_by_D?` on their truncated values `D` and `P`. This constraint is defined and proved (by `lemma3`) in $\boxed{5}$.

```
% -- Estimations                                                    5

D: VAR posrat
P: VAR rational

P_bound_by_D?(D, P): bool =
  -eps - r * rho * (D + delta) < P & P < r * rho * (D + delta)

lemma3: LEMMA
  (P <= p & p < P + eps   AND
   D <= d & d < D + delta AND
   p_over_d_bound?(d, p))
     IMPLIES
       P_bound_by_D?(D, P)
```

`lemma3` is justified by the following simple transformations, and a PVS proof of this lemma is easily obtained from these steps.

```
        -r * rho <= p / d <= r * rho
<=>       { multiplying by d }
        -r * d * rho <= p <= d * r * rho
==>       { hypothesis, d * r * rho < (D + delta) * r * rho}
        -eps - r * rho (D + delta) < P < r * rho (D + delta)
```

If the quotient digit is going to be selected on the basis of `P` and `D` rather than `p` and `d`, then the condition `legitimate?` which characterizes those values of the quotient digit `q` in terms of `p` and `q` that will yield a converging recurrence, has to be restated in terms of `P` and `D`. Inspection of the *pd-plot* in Figure 2 reveals that the legitimacy of quotient selection for the marked corners of the shaded rectangles suffices to show the legitimacy of selecting this quotient digit for all (`d`, `p`) pairs in the rectangle. If `d` and `p` satisfy `p_over_d_bound?`, then `D` and `P` satisfy `P_bound_by_D?` by `lemma3`. We need enough constraints on quotient selection in terms of `D` and `P` to guarantee that `legitimate?(q, d, p)` holds. When `p` is positive, the minimal possible value of `p / d` is `P / (D + delta)`, and the maximal value is `(P + eps) / D`, which explains the constraints in the definition of `lookup_legitimate?` for `q = a` and and `0 < q & q < a`. Similarly, when `p` is negative, the least possible value of `p / d` is `P / D` and the largest possible value is `(P + eps) / (D + delta)`. The definition when `q = 0` uses `P / D` as the lower bound on `p / d` since `p` is negative here, and `(P + eps) / D` as the upper bound since `p` is positive. Consequently, the constraint `lookup_legitimate?` in $\boxed{6}$ on lookup tables guarantees the legitimacy of quotient selection as shown in `lemma4`.

```
% -- Lookup Table                                                          6

  lookup_legitimate?(q, D, (P: rational | P_bound_by_D?(D, P))): bool =
    COND
      q = a          -> (a - rho) * (D + delta) <= P,
      0 < q & q < a  -> (q - rho) * (D + delta) <= P
                        & (P + eps) <= (q + rho)*D,
      q = 0          -> -rho * D <= P & (P + eps) <= rho * D,
     -a < q & q < 0  -> (q - rho) * D <= P
                        & (P + eps) <= (q + rho) * (D+delta),
      q = -a         -> (P + eps) <= (-a + rho) * (D + delta)
    ENDCOND

  lemma4: LEMMA
    (P <= p & p < P + eps  AND D <= d & d < D + delta AND
     p_over_d_bound?(d, p) AND lookup_legitimate?(q, D, P))
       IMPLIES legitimate?(q, d, p)
```

The proof command `(grind :theories "real_props")` combines the decision procedures of PVS with rewriting of known facts from the prelude about operations on real and rational numbers, and proves `lemma4` automatically when `r` and `a` are instantiated with specific numeric values; otherwise manual guidance is needed to deal with nonlinear equalities and inequalities.

Abstract SRT Division. We now present a schematic SRT division algorithm and prove its correctness. The basic idea underlying SRT division is that a significant reduction of the overall cycle time is obtained by computing the next partial remainder `pp(i + 1)` and predicting a quotient digit `qq(i + 1)` in parallel. An (under)estimation `PP(i)` of the next partial remainder `pp(i + 1)` is used in conjunction with an underestimated divisor `D` to estimate the quotient digit `qq(i + 1)`. Note that `PP(i)` can be computed faster than `pp(i + 1)`, since most of the time taken to compute `pp(i + 1)` is a full-precision addition, and the computation of `PP(i)` only involves a limited-precision adder.

The schematic implementation is given by assuming initially that

1. The initial value of the partial remainder `pp(0)` and the divisor `d` are within the convergent region of the recurrence, i.e., satisfy `p_over_d_bound?`.

2. The leading quotient digit `qq(0)` is somehow chosen (by preprocessing) to be a legitimate one, i.e., satisfying `legitimate?(qq(0), d, pp(0))`.

The implementation then must ensure that in each successive step, the values of the partial remainder `pp(j + 1)` and the quotient digit `qq(j + 1)` are computed from `pp(j)` and `qq(j)` so that the

1. `pp(j + 1)` satisfies `recurrence?(pp(j + 1), pp(j), qq(j), d)`, and

2. `qq(j + 1)` is chosen to satisfy the condition `lookup_legitimate?(qq(j + 1), D, PP(j))` where `PP(j)` is chosen to underestimate `pp(j + 1)` with `eps` and `D` is chosen to underestimate `d` within delta. The guarding antecedent `P_bound_by_D?(D, PP(j))` is redundant since it is an invariant, but it is used here to weaken the constraint on the possible implementations of quotient selection.

The theorem `invariant` states that the implementation guarantees the conditions

1. `p_over_d_bound?(d, pp(i))`: Each partial remainder is within the bounds needed for convergence, and

2. `legitimate?(qq(i), d, pp(i))`: The quotient digits are accurately chosen in terms of the underestimations of `pp(i)` and `d`.

```
% -- Basic Invariant                                                    7

  PP: VAR sequence[rational]

  invariant: THEOREM
        p_over_d_bound?(d, pp(0))
    AND legitimate?(qq(0), d, pp(0))
    AND (FORALL j:
               recurrence?(pp(j + 1), pp(j), qq(j), d)
           AND PP(j) <= pp(j + 1) & pp(j + 1) <  PP(j) + eps
           AND            D <= d & d < D + delta
           AND (P_bound_by_D?(D, PP(j))
                  IMPLIES lookup_legitimate?(qq(j + 1), D, PP(j))))
     IMPLIES
       (p_over_d_bound?(d, pp(i)) AND legitimate?(qq(i), d, pp(i)))

END srt
```

The proof of the invariant is by induction on the number of iterations `i`. The nontrivial part of the induction step is a chain of implications

```
     legitimate?(qq(i), d, pp(i))
       { lemma2 }
==>  p_over_d_bound?(d, pp(i + 1))
       { lemma3 }
==>  P_bound_by_D?(D, PP(i))
       { hypothesis }
==>  lookup_legitimate?(q(D, PP(i)), D, PP(i))
       { lemma4, hypothesis }
==>  legitimate?(qq(i + 1), d, pp(i + 1))
```

Altogether, to prove the correctness of a specific SRT divider circuit it suffices to show that

Step 1: The arithmetic interpretations of the computed sequences of partial remainders and quotient digits satisfy the recurrence relation `recurrence?`,

Step 2: There are constants `delta` and `eps` such that the divisor and the partial remainders are bound by under-estimators in the sense described above, and

Step 3: The quotient selection logic satisfies the `lookup_legitimate?` predicate for inputs satisfying `P_bound_by_D?`.

Whenever these conditions hold, theorem `invariant`, and consequently theorem `convergence` is applicable, and correctness follows.

6. Modeling The Data Path

The data path of an SRT division circuit with `r = 4` and `a = 2` as described by Taylor [34] is specified and proven to be correct by applying the general SRT theory developed in Section 4.

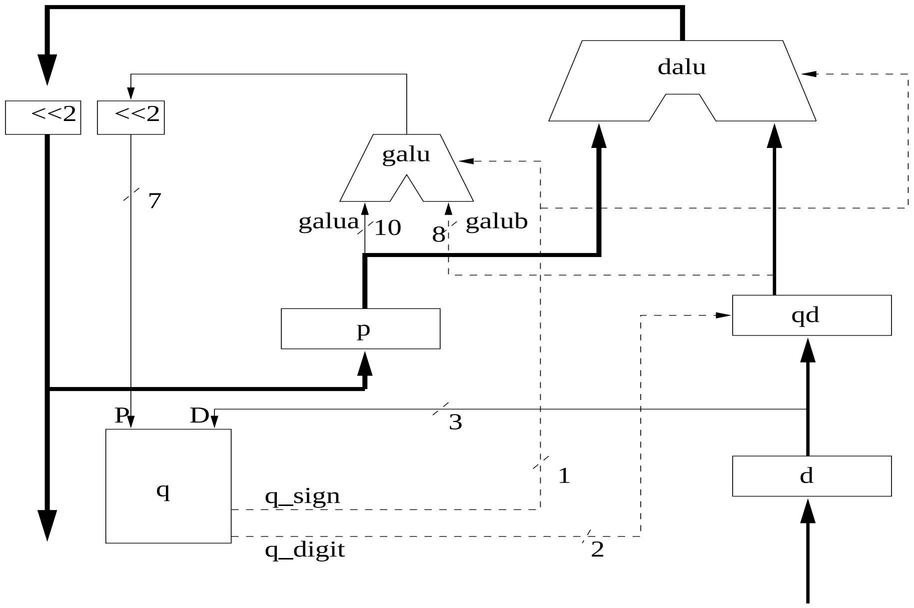

Figure 3. Data path for the SRT division circuit.

In any given iteration `i`, the circuit in Figure 3 computes a next partial remainder `p(i + 1)` and a new quotient digit `q(i + 1)` of the next partial remainder in parallel. Since `q(i + 1)` depends on `p(i + 1)` and this value is not yet available, it is necessary to compute an estimate `P(i)` of `p(i + 1)` and base the selection of `q(i + 1)` on this estimate. This estimate of the partial remainder is obtained by multiplying the output of the *guess* ALU `galu(i)` by 4 and taking the 7 leading bits. The signal `galu(i)` is obtained by supplying the leading 10 bits of `p(i)` as input `galua(i)`, and the leading 6 bits of signal `qd(i)`, which holds multiples of the divisor, as input `galub(i)`. The computation of `galu(i)` is characterized by the axioms in [19]. Essentially `galu(i)` is computed to underestimate the unshifted partial remainder `dalu(i)` by at most 2^{-6} (i.e., `ulp(6)`). It does this by adding `galua(i)` and `galub(i)` when the previous quotient digit sign `q_sign(i)` is negative, and subtracting `(galub(i) + ulp(6))` from `galua(i)` when `q_sign(i)` is negative. This asymmetry in the computation of `galu(i)` ensures that `P(i)` underestimates `p(i+1)` by at most 3/16. This 7-bit estimation `P(i)` together with a three bit truncation of the divisor `d(i)`[3] serve as indices to a lookup table `lookup` (shown in Appendix A.2). The result of this lookup is split into a value and a sign part, and is delayed for one time unit. Altogether, the computation of the estimation and the selection of the quotient digit is computed faster than the actual next partial remainder `p(i + 1)`, since most of the time taken to compute `p(i + 1)` is a full-precision addition.

The signals of this circuit are described by uninterpreted sequences indexed over time. The type `time` is simply defined as `time : TYPE = nat` in theory `time`, and signals are represented by the type `signal` which is isomorphic to sequences over `time` (see Appendix A.1).

The signals of the circuit in Figure 3 are declared as uninterpreted constants that are signals of bit-vectors of various fixed lengths, and the uninterpreted constant `N`, where `N > 8`, determines the width of the data paths for the divisor and the partial remainders; examples of signal declarations are listed in 8 and the complete structural specification of the SRT circuit is given in Appendix A.3.

```
d: signal[bvec[N]]                                                    8
P: signal[bvec[7]]
```

Next, arithmetic interpretations of the bit-vector signals are assigned to the signals of the circuit.

```
d(i): rational = fp[1, N - 1].val(d(i))                               9
P(i): rational = fp2c[4, 3].val(P(i))
```

The divisor bit-vector `d(i)`, for example, is interpreted as an unsigned and normalized fixed-point bit-vector, and the output of the guess ALU `galu(i)` is interpreted as a 2's-complement fixed-point bit-vector with two integral bits and `6` residual bits. The complete set of signal interpretations is listed in Appendix A.3 and the theories `fp` and `fp2c` are stated in Appendix A.1. Note also that overloading the name of the bit-vector signal with its arithmetic interpretation mimics a specification style often found in textbooks about computer arithmetic.

The datapath specification differs slightly from that of Taylor [34] where he implicitly, i.e., without explanation, uses an optimization in which the two leading bits (the sign bit and the most significant integer bit) of the partial remainder inputs to the GALU and the DALU are discarded. This optimization is justified since these bits do not contribute to the output of these ALUs. Since a proof of this optimization is outside the scope of the paper, we have retained these two bits in our datapath. The datapath widths and their arithmetic interpretations have to be carefully specified since it is easy to obtain an unimplementable, i.e., inconsistent, collection of axioms.

The inputs to the quotient selection unit `lookup` are the three bit truncation `D` of the divisor `d` and the seven bit approximation `P` of the next partial remainder.

```
 lookup((D: bvec[3]),                                                 10
        (P: bvec[7] |
           P_bound_by_D?(1 + fp[0,3].val(D), fp2c[4, 3].val(P))))
 :{ q: subrange(-2, 2) |
     lookup_legitimate?(q, 1 + fp[0,3].val(D), fp2c[4, 3].val(P)) }
```

In 10, predicate subtypes serve as a specification of a set of quotient lookup tables by means of domain and range constraints, and a specific implementation of these constraints is proven correct in Section 7. Note that the type shown in 10 is meant to be illustrative and is not part of the specification where it is proved as a *typing judgement* for the actual lookup table in Appendix A.2.

Finally, the behavior of the data path is specified by the equalities and inequalities given as axioms in Appendix A.3.

From these formalizations, the `grind` strategy proves the lemmas in 11 about Taylor's division circuit in Figure 3.

```
taylor_lemma1: LEMMA recurrence?(p(i + 1), p(i), q(i), d(i))                11

taylor_lemma2: LEMMA galu(i) <= dalu(i) & dalu(i) < galu(i) + 2 * ulp(6)

taylor_lemma3: LEMMA P(i) <= p(i + 1) & p(i + 1) < P(i) + 3/16
```

Together with the constraint on `D` with respect to `d` in Appendix A.3, this accomplishes Steps 1 and 2 with `delta = 1/8` and `eps = 3/16` mentioned in Section 5. Now it is a simple matter of instantiating theorems `invariant` in 7 and `convergence` in 3 to obtain the invariant results in 12 for this specific circuit.

```
taylor_invariant: LEMMA                                                      12
  p_over_d_bound?(d(0), p(i)) AND legitimate?(q(i), d(0), p(i))

taylor_convergence: THEOREM
   LET residue =  p(0) / d(0) - val(i + 1, q) IN
     - 2 / (3 * 4^i) <= residue & residue <= 2 / (3 * 4^i)
```

7. The Lookup Table

The legitimacy constraint `lookup_legitimate?` (see Section 4) on quotient lookup tables permits different implementations, and Taylor [34] develops a particularly compact one. This table computes the next quotient digit from the truncation `D:bvec[3]` of the divisor to the three leading bits and the estimation `P:bvec[7]` of the next partial remainder. Bits 6 down to 2 of `P` are used as a table index and the remaining bits are used in some cases to compute the resulting value.

The formalization of the resulting table in Appendix A.2 uses the `TABLE` construct of the PVS specification language [24]. This construct was added to the PVS specification language in order to provide visually appealing two-dimensional tabular specifications in the manner advocated by Parnas and others [26]. It proved adequate to express the lookup table of this SRT circuit in a concise and perspicuous way. In particular, blank entries in the lookup table in Appendix A.2 cause the type-checker to generate TCCs which ensure that viable arguments `D`, `P` never point to such a blank entry. Furthermore, the table construct requires that the lookup is functional and ensures this by generating *disjointness* and *coverage* TCCs. From 13 one concludes that the given table `lookup` indeed satisfies the constraint given for lookup tables in Section 6. The proof follows a case analysis on each of the entries in the table using the type constraints on `D` and `P`. In the proof in 13, five interactions with PVS are needed to check all but one of the table entries, and five further interaction are needed to deal with the one entry a that involves nonlinear arithmetic. The expansion of P into its bit-level representation has to be carefully controlled or this proof can become time-consuming due to an inefficiency in the PVS decision procedures. The type correctness conditions generated by the type-checker for the lookup table are proven with similar strategies.

```
(FORALL (D, (P: bvec[7] | P_bound_by_D?(valD(D), valP(P)))):          13
  lookup_legitimate?(lookup(D, P), valD(D), valP(P)))
```

In the course of proving the consistency of the lookup table in Appendix A.2, PVS has proven helpful as a debugging tool and came up with precise *counterexamples*.[4] By injecting, for example, a wrong value 0 at a certain position in the lookup table in Appendix A.2 and rerunning the proof above, the PVS system returns with the following subgoal (slightly edited for purposes of presentation) it could not solve.

```
 P!1(6) = 1, P!1(5) = 1, P!1(4) = 0, P!1(3) = 0, P!1(2) = 0             14
 P_bound_by_D?(...)
 D!1(2) = 0, D!1(1) = 1, D!1(0) = 0
|-------
 lookup_legitimate?(...)
```

This unsolved subgoal provides the counterexample immediately and points to the altered table entry at position Dp = 2 and Ptruncp = -8.

8. Conclusions

We have shown how PVS can be used to specify and prove the correctness of a non-trivial SRT division algorithm and its hardware implementation in a modular way.

The theory underlying SRT dividers is formalized by introducing the basic arithmetic definitions and facts about non-restoring division. This formalization is used to derive constraints that a specific circuit must satisfy in order to correctly implement the required recurrence relation while ensuring that each partial remainder stays below a certain threshold. This step abstracts away from any particular implementation of the lookup table, while formulating the constraints on a lookup table that are sufficient for the overall correctness of the algorithm. Finally, we demonstrate the correctness of a specific lookup table implementation.

This modular approach not only structures the specifications and the proof in a nice way but also has the advantage that slight variations of this particular circuit design and lookup table can be verified by redoing just one part of the proof. Moreover, parts of the theory can be reused for verifying similar division, and perhaps even square-root, circuits.

This verification exercise demonstrates the value of efficient decision procedures and the use of an expressive specification language in mechanized verification. The concepts of predicate subtypes, overloading, and tables of the PVS specification language proved to be very useful for expressing the high-level designs of this arithmetic circuit in a concise and natural way. Such high-level descriptions not only minimize the pitfalls of introducing errors in initial design specification but also open the door to using these specifications as design documents. Moreover, the tight integration of decision procedures with rewriting strategies of PVS proved to be a useful workhorse, since the circuit specific theorems and the correctness of the table implementation are proven in a fairly automatic way. In proof obligations that involve nonlinear equalities, however, the PVS prover typically must be manually guided to construct the proofs.

The correctness proof of the table implementation in Section 7 takes around 10 minutes on a Sparc-20 and 3 minutes on an Ultrasparc-2. The entire proof consisting of 85 lemmas and TCC proof obligations can be checked in around 4500 seconds of wallclock time on a Sparc-20.

Acknowledgements. We are grateful to Ed Clarke and Steven German who pointed us to Taylor's paper [34] on SRT division and challenged us to verify Taylor's algorithm and lookup table in PVS. We also thank the anonymous referee for providing detailed suggestions for improving the paper, to John Rushby and Friedrich von Henke for their encouragement during this work, and to Sam Owre for help with PVS and for his careful reading of drafts of the paper. We were also helped by Paul Miner and Jim Leathrum's PVS proofs in improving the efficiency with which the lookup table and related proof obligations were analyzed.

Notes

1. A manuscript describing this work can be obtained from the URL `www.ece.utexas.edu/~lynch/divide/divide.html`.
2. The URL `www.csl.sri.com/pvs/examples/SRT` contains the relevant PVS specifications and proofs.
3. Actually four bits, but the divisor `d(i)` is assumed to be normalized; therefore the first bit of `d(i)` is always `1`.
4. Even though the original design of Taylor's lookup table in [34] proved to be correct, we still managed to accidentally inject errors in the initial PVS transcriptions.

References

1. Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.
2. D.E. Atkins. Higher-radix Division Using Estimates of the Divisor and Partial Remainders. *IEEE Transactions on Computers*, C-17(10):925–934, October 1968.
3. Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the 32nd Design Automation Conference*, pages 535–541, San Francisco, CA, June 1996.
4. W. W. Bledsoe. The SUP-INF method in Presburger arithmetic. Technical Report Memo ATP-18, The University of Texas at Austin, Math Department, December 1974.
5. Randal E. Bryant. Bit-level analysis of an SRT divider circuit. In *Proceedings of the 33rd Design Automation Conference*, pages 661–665, Las Vegas, NV, June 1996.
6. Yirng-An Chen, Edmund Clarke, Pei-Hsin Ho, Yatin Hoskote, Timothy Kam, Manpreet Khaira, John O'Leary, and Xudong Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In Srivas and Camilleri [32], pages 19–33.

7. E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams: Overcoming the limitations of MTBDDs and BMDs. Technical Report CMU-CS-95-159, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, April 1995.
8. E.M. Clarke and S.M. German. Personal Communication, 1995.
9. E. M. Clarke, S. M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In Alur and Henzinger [1], pages 111–122.
10. Shiu-Kai Chin. Verified functions for generating signed-binary arithmetic hardware. *IEEE Transactions on Computer-Aided Design*, 11(12):1529–1558, August 1992.
11. E. M. Clarke, M. Khaira, and X. Zhao. Word level symbolic model checking—Avoiding the Pentium FDIV error. In *Proceedings of the 33rd Design Automation Conference*, pages 645–648, Las Veqas, NV, June 1996.
12. D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer-Verlag.
13. S.M. German. Towards Automatic Verification of Arithmetic Hardware. Lecture notes, March 1995.
14. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
15. Standard for Binary Floating-Point Arithmetic, 1985. ANSI/IEEE Std 754-1985.
16. Standard for Radix-Independent Floating-Point Arithmetic, 1987. ANSI/IEEE Std 854-1987.
17. Deepak Kapur and M. Subramaniam. Mechanically verifying a family of multiplier circuits. In Alur and Henzinger [1], pages 135–146.
18. Gila Kamhi, Osnat Weissberg, Limor Fix, Ziv Binyamini, and Ze'ev Shtadler. Automatic datapath extraction for efficient usage of HDDs. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 95–106, Haifa, Israel, June 1997. Springer-Verlag.
19. M. Leeser and J. O'Leary. Verification of a Subtractive Radix-2 Square Root Algorithm and Implementation. In *Proc. of ICCD'95*, pages 526–531. IEEE Computer Society Press, 1995.
20. O.L. McSorley. High-speed Arithmetic in Binary Computers. In *Proc. of IRE*, pages 67–91, 1961.
21. Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Srivas and Camilleri [32], pages 64–78.
22. J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5_K86 floating-point division algorithm. *IEEE Transactions on Computers*, 1997. To appear.
23. S.F. Oberman and M.J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, Feb. 1997.
24. Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.
25. S. Owre, J. Rushby, N. Shankar, and F. 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.
26. D. L. Parnas. Using mathematical models in the inspection of critical software. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, International Series in Computer Science, chapter 2, pages 17–31. Prentice Hall, 1995.
27. 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*, number 915 in Lecture Notes in Computer Science, pages 97–107. Springer Verlag, May 1995.
28. J.E. Robertson. A new Class of Digital Division Methods. In *IRE Trans. on Electron. Computers*, volume EC-7, pages 218–222, 1958.
29. H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Alur and Henzinger [1], pages 123–134.
30. Harald Rueß. Hierarchical verification of two-dimensional high-speed multiplication in PVS: A case study. In Srivas and Camilleri [32], pages 79–93.

31. David M. Russinoff. A mechanically checked proof of the correctness of the AMD K5 floating-point square root algorithm. This Journal.
32. Mandayam Srivas and Albert Camilleri, editors. *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1996. Springer-Verlag.
33. Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.
34. G.S. Taylor. Compatible Hardware For Division and Square Root. In *Proceedings of the 5th Symposium on Computer Arithmetic*, pages 127–134. IEEE Computer Society Press, 1981.
35. K.D. Tochter. Techniques of Multiplication and Division for Automatic Binary Computers. In *Quart. J. Mech. Appl. Match*, volume Part 3, pages 364–384, 1958.
36. 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*, 3:5–31, January 1994.

Appendix A

A.1. Miscellaneous

Time

```
time: THEORY
BEGIN

 time: TYPE = nat

END time
```

Signals

```
signal[T: TYPE]: THEORY
BEGIN
  IMPORTING time

  signal: TYPE = [time -> T]

END signal
```

Unit of Least Position

```
ulp: THEORY

BEGIN

  ulp(m: nat): posrat = 2^(-m)

END ulp
```

Fixed-Point Interpretation I Fixed-Point interpretation of binary bitvectors with k integral and m residual bits.

```
fp[k, m: nat]: THEORY

BEGIN

  IMPORTING bitvectors@bv_nat

  val(bv: bvec[k + m]): nonneg_rat  = 2^(-m) * bv2nat(bv)

END fp
```

Fixed-Point Interpretation II 2's-complement binary floating-point with k integral and m residual bits.

```
fp2c[k: posnat, m: nat]: THEORY

BEGIN

  IMPORTING bitvectors@bv, bitvectors@bv_int

  val(bv: bvec[k + m]): rational =
    2^(-m) * (-2^(k + m - 1) * bv(k + m - 1) + bv2nat_rec(k + m - 1, bv))

END fp2c
```

A.2. Implementation of the Lookup Table

```
val: THEORY

  BEGIN

   IMPORTING bitvectors@bv, fp[0, 3], fp2c[4, 3]

   valD(D: bvec[3]): posrat   = 1 + fp[0,3].val(D)
   valP(P: bvec[7]): rational = fp2c[4, 3].val(P)

  END val
```

```
misc: THEORY

BEGIN

  IMPORTING bitvectors@bit

  bool_chain: LEMMA (FORALL (b: bit): NOT b = 0 IMPLIES b = 1)

END misc
```

```
lookup: THEORY

BEGIN

  IMPORTING bitvectors@bv, misc, val,
            srt[4, 2, 1/8, 3/16]

  D: VAR bvec[3]
  P: VAR bvec[7]

  valD_eq: LEMMA
    valD(D) = 1 + 1/2 * D(2) + 1/4 * D(1) + 1/8 * D(0)

  valP5(P) : int = -16 * P(6) + 8 * P(5) + 4 * P(4) + 2 * P(3)
                   + P(2)

  valP_eq: LEMMA
    valP(P) = 1/2 * valP5(P) + 1/4 * P(1) + 1/8 * P(0)

 lookup(D, (P: bvec[7] | P_bound_by_D?(valD(D), valP(P))))
     : subrange(-2, 2) =
   LET
        a = -(2 - P(1) * P(0)),
        b = -(2 - P(1)),
        c = 1 + P(1),
        d = -(1 - P(1)),
        e = P(1),
        Dp      = 8 * valD(D) - 8,
        Ptruncp = valP5(P)
    IN
    [15]

  lookup_legitimate: THEOREM
   (FORALL (D, (P: bvec[7] | P_bound_by_D?(valD(D), valP(P)))):
      lookup_legitimate?(lookup(D, P), valD(D), valP(P)))

  JUDGEMENT lookup HAS_TYPE
    [D: bvec[3], P: P: bvec[7] | P_bound_by_D?(valD(D), valP(P))
       -> {q: subrange(-2, 2) | lookup_legitimate?(q, valD(D), valP(P))}]

END lookup
```

15

```
TABLE Ptruncp, Dp
      |[   0|   1|   2|   3|   4|   5|   6|   7]|
 %---------------------------------------------%
 |  10 |    |    |    |    |    |    |    |  2 ||
 |   9 |    |    |    |    |    |  2 |  2 |  2 ||
 |   8 |    |    |    |    |  2 |  2 |  2 |  2 ||
 |   7 |    |    |  2 |  2 |  2 |  2 |  2 |  2 ||
 |   6 |    |  2 |  2 |  2 |  2 |  2 |  2 |  2 ||
 |   5 |  2 |  2 |  2 |  2 |  2 |  2 |  2 |  1 ||
 |   4 |  2 |  2 |  2 |  2 |  c |  1 |  1 |  1 ||
 |   3 |  2 |  c |  1 |  1 |  1 |  1 |  1 |  1 ||
 |   2 |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 ||
 |   1 |  1 |  1 |  1 |  1 |  e |  0 |  0 |  0 ||
 |   0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
 |  -1 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
 |  -2 | -1 | -1 |  d |  d |  0 |  0 |  0 |  0 ||
 |  -3 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 ||
 |  -4 |  a |  b | -1 | -1 | -1 | -1 | -1 | -1 ||
 |  -5 | -2 | -2 | -2 |  b | -1 | -1 | -1 | -1 ||
 |  -6 | -2 | -2 | -2 | -2 | -2 | -2 |  b | -1 ||
 |  -7 | -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 ||
 |  -8 |    |    | -2 | -2 | -2 | -2 | -2 | -2 ||
 |  -9 |    |    |    | -2 | -2 | -2 | -2 | -2 ||
 | -10 |    |    |    |    |    | -2 | -2 | -2 ||
 | -11 |    |    |    |    |    |    | -2 | -2 ||
 %---------------------------------------------%
 ENDTABLE
```

A.3. Specification of the Datapath

Taylor's radix-4 SRT division circuit with quotient prediction.

Header

```
Taylor[N: upfrom(8)]: THEORY

BEGIN

   IMPORTING time, signal, bitvectors@bv

   i: VAR time

   delta: posrat = 1/8   % Circuit Specific Constants
   eps  : posrat = 3/16

   IMPORTING srt[4, 2, 1/8, 3/16], fp, fp2c
   ...
```

Structural Specification

```
d        : signal[bvec[N]]      % Divisor
qd       : signal[bvec[N + 1]]  % Multiples of divisor
p        : signal[bvec[N + 3]]  % Register for partial remainders
dalu     : signal[bvec[N + 1]]  % Computation of new partial remainder
galua    : signal[bvec[10]]     % Truncated Partial Remainder
galub    : signal[bvec[8]]      % Truncated Divisor
galu     : signal[bvec[8]]      % Estimated dalu-output
P        : signal[bvec[7]]      % Estimation of remainder
D        : signal[bvec[3]]      % Truncated Divisor
q_sign   : signal[bit]          % Sign of quotient digit
q_digit : signal[{ bv: bvec[2] | bv(0) = 0 OR bv(1) = 0 }]
     % Absolute value of quotient digit
```

Arithmetic Interpretations

```
d(i)       : rational = fp[1, N - 1].val(d(i))                     16
D(i)       : posrat   = 1 + fp[0, 3].val(D(i))
qd(i)      : rational = fp[2, N - 1].val(qd(i))
p(i)       : rational = fp2c[4, N - 1].val(p(i))
dalu(i)    : rational = fp2c[2, N - 1].val(dalu(i))
galua(i)   : rational = fp2c[4, 6].val(galua(i))
galub(i)   : rational = fp[2, 6].val(galub(i))
galu(i)    : rational = fp2c[2, 6].val(galu(i))
P(i)       : rational = fp2c[4, 3].val(P(i))
q_digit(i): upto[2]   = fp[2, 0].val(q_digit(i))
sign(i)    : int      = IF q_sign(i) = 1 THEN 1 ELSE -1 ENDIF
q(i)       : subrange[-2, 2] = q_digit(i) * sign(i)
```

Initial Behavioral Specification

```
d_bound: AXIOM 1 <= d(i)                                           17
q_init : AXIOM legitimate?(q(0), d(0), p(0))
p_init : AXIOM p_over_d_bound?(d(0), p(0))

JUDGEMENT d HAS_TYPE signal[posrat]
```

Behavioral Specification (Equalities)

```
  IMPORTING ulp                                                        18

  d_eqn   : AXIOM  d(i + 1) = d(i): posrat
  D_eqn   : AXIOM  D(i + 1) = D(i): posrat
  qd_eqn  : AXIOM     qd(i) = q_digit(i) * d(i)
  p_eqn   : AXIOM  p(i + 1) = 4 * dalu(i)

  dalu_eqn: AXIOM
   dalu(i) = IF sign(i) = 1 THEN p(i) - qd(i) ELSE p(i) + qd(i) ENDIF

  galu_eqn: AXIOM
   galu(i) = IF sign(i) = 1 THEN galua(i) - galub(i) - ulp(6)
                            ELSE galua(i) + galub(i) ENDIF
```

Behavioral Specification (Inequalities)

```
  galua_ineq  : AXIOM  galua(i) <= p(i) & p(i) < galua(i) + ulp(6)     19
  galub_ineq  : AXIOM  galub(i) <= qd(i) & qd(i) < galub(i) + ulp(6)
  galu_ineq   : AXIOM  P(i) <= 4 * galu(i)
                       & 4 * galu(i) <= P(i) + ulp(3) - ulp(4)
  d_bound_by_D: AXIOM  D(i) <= d(i) & d(i) < D(i) + 1/8
```

Auxiliary Lemmas, Recurrence, and Estimation Errors

```
 % -- Auxiliary Lemmas

  d_char: LEMMA d(0) = d(i): posrat
  D_char: LEMMA D(0) = D(i): posrat


 % -- Basic Recurrence of SRT-Loop

  taylor_lemma1: LEMMA
   recurrence?(p(i + 1), p(i), q(i), d(0))


 % -- Maximal Error of Estimation

  taylor_lemma2: LEMMA
    galu(i) <= dalu(i) & dalu(i) < galu(i) + 2 * ulp(6)

  taylor_lemma3: LEMMA
    P(i) <= p(i + 1) & p(i + 1) < P(i) + eps
```

Behavioral Specification (Lookup)

```
   IMPORTING lookup

   q_step: AXIOM
    P_bound_by_D?(D(i), P(i)) IMPLIES q(i + 1) = lookup(D(i), P(i))
```

Correctness of the Circuit

```
   taylor_invariant: LEMMA
    p_over_d_bound?(d(0), p(i)) AND legitimate?(q(i), d(0), p(i))

   taylor_convergence: THEOREM
     LET residue =  p(0) / d(0) - val(i + 1, q) IN
      - 2 / (3 * 4^i) <= residue & residue <= 2 / (3 * 4^i)

END Taylor
```