Formal Methods in System Design 14, 7–44 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

# Verifying the SRT Division Algorithm Using Theorem Proving Techniques

EDMUND M. CLARKE emc@cs.cmu.edu
*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

STEVEN M. GERMAN german@watson.ibm.com
*IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, USA*

XUDONG ZHAO xzhao@cs.cum.edu
*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

**Abstract.** We verify the correctness of an SRT division circuit similar to the one in the Intel Pentium processor. The circuit and its correctness conditions are formalized as a set of algebraic relations on the real numbers. The main obstacle to applying theorem proving techniques for hardware verification is the need for detailed user guidance of proofs. We overcome the need for detailed proof guidance in this example by using a powerful theorem prover called Analytica. Analytica uses symbolic algebra techniques to carry out the proofs in this paper with much less guidance than existing general purpose theorem provers require for algebraic reasoning.

**Keywords:**

## 1. Introduction

Proving the correctness of arithmetic operations has always been an important problem. The importance of this problem has been recently underscored by the highly publicized division error in the Pentium processor [20]. Some people have estimated that this error cost Intel almost 500 million dollars [2]. In this paper, we verify a division circuit [22] that is similar to the one used in the Pentium. The circuit uses a radix four SRT division algorithm that looks ahead to find the next quotient digit in parallel with the generation of next partial remainder. An 8-bit ALU estimates the next remainder's leading bits. A quotient digit look-up table generates the next quotient digit depending on the leading bits of the estimated remainder and the leading bits of the divisor.

In our approach to verification, we formalize the circuit and its correctness conditions as a set of algebraic relations over the real numbers [12]. These algebraic relations correspond closely to the bit-level structure of the circuit, and could have been generated mechanically from a hardware description. Most of the hardware for the SRT algorithm can be described by linear inequalities. This led us to experiments [12] in which we proved properties of the SRT hardware using a special purpose program for verifying SRT dividers. The program was written in the Maple symbolic algebra system and used its Simplex algorithm package.

We now have a more general yet highly automatic approach, where the correctness of the circuit is proved using a powerful theorem prover called Analytica [10] that we have developed. Analytica is the first theorem prover to use symbolic computation techniques in a major way. It is written in the Mathematica programming language and runs in the interactive environment provided by this system [24]. Compared to Analytica, most theorem provers require significant user interaction when proving results with algebraic content. The main problem is the large amount of domain knowledge that is required for even the simplest proofs. Our theorem prover, on the other hand, is able to exploit the mathematical knowledge that is built into the symbolic computation system to carry out long algebraic computations automatically. For example, the proof of the main correctness result for the SRT divider requires two proof steps in Analytica: first step proves a lemma, and the second step proves the main result by applying the lemma. The entire input to Analytica for the SRT problem, including a mathematical model of the circuit and the specifications of four theorems to be proved, is only 50 lines of text.

The previous work that is most closely related to ours is by Verkest et al. [23], who have verified a nonrestoring division algorithm and hardware implementation using the Boyer Moore theorem prover [6]. The circuit they consider is much simpler than the one we verify. The main difficulty in verifying our circuit is in showing that the estimation circuit and the quotient lookup table give the correct quotient digits. In contrast, their circuit computes the quotient in radix 2, and does not speed up the computation by estimating the partial remainders. Another project by Leeser et al. [15] verifies a radix 2 square root algorithm and hardware implementation. This work is similar to [23] and does not involve the design features that make fast division circuits difficult to verify. Although we prove the correctness of a relatively complicated circuit, our use of symbolic computation techniques allows us to carry out the proof automatically.

After our initial experiment in verifying the SRT circuit using the Maple symbolic algebra system, we communicated our results with the Maple system to researchers at SRI International [7], who then proceeded to verify the same circuit using the PVS theorem prover [18]. Although there are many differences between our approach and the SRI approach, the SRI researchers have retained some of the ideas of our mathematical model of the circuit, such as modelling truncation of arithmetic signals using linear inequalities [12] (cf. Section 4.1). One important difference between the approaches is that the SRI group verified a parameterized version of the division circuit, while we have developed a method that can verify individual instances of the division circuit highly automatically. The proof of the parameterized division circuit in [18] requires much more user guidance than is needed with our method.

An earlier version of this paper appeared in [8]. The paper is organized as follows: In Section 2 we describe a circuit that implements a radix four SRT division algorithm. In Section 3 we give an overview of the theorem prover Analytica that we use for verifying the circuit. In particular, we explain how inequalities are handled. Section 4 is the heart of the paper. It contains the axioms that specify the behavior of the circuit and the theorems about the circuit that we have proved using Analytica. The paper concludes in Section 5 with a summary of our work and some directions of future research. In the Appendix, we show the input to the theorem prover and part of the generated proof.

## 2. The SRT division algorithm and circuit

### 2.1. *Floating-point numbers and floating division*

Under the IEEE arithmetic standard, a normalized floating point number has the form *sign* · *significand* · $2^{exponent}$, where *sign* is $\pm 1$, represented by one bit, the *significand* is a rational number in the range $1 \leq significand < 2$, and *exponent* is an integer. Certain values, such as 0, have special representations under the standard. Hardware circuits for floating-point arithmetic are usually organized into two parts: a normalization circuit and an arithmetic core, which performs arithmetic operations on the significands of the normalized numbers. The circuit that we consider in this paper is the core of a floating point division circuit. A separate circuit handles the signs and exponents.

There are several ways to interpret the arithmetic operation performed by the hardware of the core. One way is to consider it as an operation on scaled integers. In this paper, we interpret signals in the division core as arbitrary rational numbers, and develop our proof using algebraic theory that holds for all the rationals, not just the values that can be represented in a certain number of bits. One advantage of our approach is that our specification and correctness proof are independent of the hardware word length; that is, we prove the correctness of the SRT division circuit for all word lengths $n \geq 10$ bits, *without* having to induct on word length. Note that this approach is sound but may not yield a proof in all cases. It is possible, for example, to design a floating-point circuit whose correctness depends on the fact that only a finite set of values is represented.

### 2.2. *Long division*

We begin by recalling the traditional algorithm for long division. The *Dividend* is a non-negative rational and the *Divisor* is a positive rational. The division algorithm computes the quotient as a sequence of digits in a given number radix, $r$. We assume that the inputs are such that the quotient, *Dividend*/*Divisor*, is less than $r$. In this case, the quotient will have the form $q_0 \cdot q_1 \cdots q_{m-1}$, where each of the $q_i$ is a radix $r$ digit. In order to compute the quotient digits, the algorithm computes a sequence of *partial remainders* $p_i$ according to the following recurrence:

$$p_0 = Dividend,$$
$$p_{j+1} = r \cdot (p_j - q_j \cdot Divisor), \quad \text{for } j = 0, \ldots, m-1 \tag{1}$$

Each quotient digit $q_i$ is an integer in the range $0 \leq q_j \leq r - 1$. The quotient digit $q_j$ is chosen so that

$$0 \leq p_j - q_j \cdot Divisor < Divisor. \tag{2}$$

That is, the algorithm subtracts a multiple of the divisor from the partial remainder at each step; the digit $q_j$ is chosen so that $q_j \cdot Divisor$ is the largest multiple of *Divisor* that can be subtracted from $p_j$ and still give a non-negative result. Note that in the recurrence (1), multiplying the quantity $p_j - q_j \cdot \mathit{Divisor}$ by $r$ shifts the partial remainder one digit to the left, to prepare for the next cycle of the division.

*Example 1.* As an illustration, consider using the above recurrence to compute $1/8$ to an accuracy of three decimal digits to the right of the decimal point.

$$\begin{aligned} \mathit{Dividend} &= 1.000 \\ \mathit{Divisor} &= 8.000 \end{aligned}$$

*Step 1.* Choose $q_0$ so that $0 \le p_0 - q_0 \cdot \mathit{Divisor} < 8.000$. Substituting 1.000 for $p_0$ and 8.000 for *Divisor*, this is $0 \le 1.000 - q_0 \cdot 8.000 < 8.000$. Select the quotient digit $q_0 = 0$. Then compute $p_1$:

$$\begin{aligned} p_1 &= 10 \cdot (1.000 - 0 \cdot 8.000) \\ &= 10 \cdot (1.000 - 0) \\ &= 10 \cdot 1.000 \\ &= 10.000 \end{aligned}$$

*Step 2.* Choose $q_1$ so that $0 \le 10.000 - q_1 \cdot 8.000 < 8.000$. This gives $q_1 = 1$.

$$\begin{aligned} p_2 &= 10 \cdot (10.000 - 1 \cdot 8.000) \\ &= 10 \cdot (10.000 - 8.000) \\ &= 10 \cdot 2.000 \\ &= 20.000 \end{aligned}$$

*Step 3.* Choose $q_2$ so that $0 \le 20.000 - q_2 \cdot 8.000 < 8.000$. This gives $q_2 = 2$.

$$\begin{aligned} p_3 &= 10 \cdot (20.000 - 2 \cdot 8.000) \\ &= 10 \cdot (20.000 - 16.000) \\ &= 10 \cdot 4.000 \\ &= 40.000 \end{aligned}$$

*Step 4.* Choose $q_3$ so that $0 \le 40.000 - q_3 \cdot 8.000 < 8.000$. This gives $q_3 = 5$.

$$\begin{aligned} p_4 &= 10 \cdot (40.000 - 5 \cdot 8.000) \\ &= 10 \cdot (40.000 - 40.000) \\ &= 10 \cdot 0.000 \\ &= 0.000 \end{aligned}$$

The final quotient is 0.125 and the remainder is $p_4 = 0.000$.

Observe that in the example, we were able to choose quotient digits at each step to make the quantity $p_j - q_j \cdot \mathit{Divisor}$ satisfy inequality 2. Since $p_{j+1} = r \cdot (p_j - q_j \cdot \mathit{Divisor})$, $p_j$ remained in the range

$$0 \leq p_j < r \cdot \mathit{Divisor}. \tag{3}$$

In fact, this inequality is an invariant of this division algorithm for all proper inputs. The property that the partial remainders stay in a fixed range as $j$ increases is needed to establish that the computation converges to the correct quotient. Informally, the reason the quotient computation converges is as follows: On each step of the algorithm, the partial remainder is effectively shifted one digit to the left by multipling it by $r$. One can see that the "actual" remainder of the division corresponds to the unshifted value of $p_j$. Thus, if $p_j$ stays in a fixed range, the actual remainder becomes smaller on each cycle. We develop this argument formally in Section 4.2.

It is straightforward to show that the inequality (3) is an invariant of the division algorithm. For the case $j = 0$, recall that $p_0$ is assigned the value *Dividend*. Since both *Dividend* and *Divisor* are the significands of normalized numbers, they are in the range $\{x : 1 \leq x < r\}$. It follows that (3) holds for $p_0$. For the inductive case, the algorithm chooses $q_j$ so that $0 \leq q_j \leq r - 1$ and $0 \leq p_j - q_j \cdot \mathit{Divisor} < \mathit{Divisor}$. If (3) holds for $p_j$, there must be a value of $q_j$ that satisfies these inequalities. Thus, for $p_{j+1} = r \cdot (p_j - q_j \cdot \mathit{Divisor})$, we have $0 \leq p_{j+1} < r \cdot \mathit{Divisor}$. Inequalities such as (3) are important in understanding SRT division, and we will consider them again in the next section.

## 2.3. *SRT division*

The motivation for the SRT algorithm is to provide a faster division method. The running time of the traditional long division algorithm discussed above depends on the number of iterations of (1) and the time needed for each iteration. The number of iterations needed to compute the quotient to a given number of bits $b$ of accuracy depends on the radix $r$. If the quotient is represented in radix 2, $b$ iterations will be needed, because each iteration produces only one bit of the quotient.

In practice, radix 4 is often used in hardware division circuits because only $b/2$ iterations are needed and the calculations on each iteration can be performed quickly in hardware. Each iteration involves two multiplications and a subtraction, assuming $q_j$ is known. In radix $r = 4$, both of the multiplications can be implemented by fast hardware that simply shifts one of the operands to the left. For example, the multiplication by $r$ can be computed by shifting two bits to the left. Also, the multiplication by $q_j$ can be done by simple logical operations when the value of $q_j$ is 0, or 1; i.e., select all zeroes for $0 \cdot \mathit{Divisor}$; select *Divisor* for $1 \cdot \mathit{Divisor}$. The value $2 \cdot \mathit{Divisor}$ can be generated by shifting one bit to the left. In the case that $q_j = 3$, there is a potential problem because multiplication by 3 is more difficult. We will see, however, that the SRT algorithm uses a representation of the quotient digits that avoids this problem.

The subtraction operation in (1) dominates the time needed for each iteration. For double precision arguments, a 64 bit subtraction must be performed on each cycle. The basic idea of the SRT algorithm is to arrange the computation so that the quotient digit selection can be done in parallel with the long subtraction operation. Referring to the basic recurrence (1), it is clear that the choice of $q_j$ depends on the value of $p_j$.

In order to carry out quotient selection concurrently with the computation of $p_j$, the SRT algorithm allows the choice of the quotient digit at each step to be *inexact*. SRT division computes an estimate of $p_j$ while the full subtraction is in progress. The estimated value of $p_j$ is used to select a quotient digit, but the estimate is not precise enough to guarantee that the exact quotient digit will be selected. This differs from the long division of Section 2.2, where there was only one possible choice of quotient digit at each step.

The SRT division algorithm uses the basic recurrence (1), but with different ranges for the partial remainders and quotient digits. To accomodate inexactly chosen quotient digits, SRT division allows the partial remainders to occupy a symmetric range about zero, $-k \cdot Divisor \leq p_j \leq k \cdot Divisor$, where $k$ is a constant. Also, the quotient digits are selected from a symmetric range $-a \leq q_j \leq a$, where $a$, the *quotient digit bound*, is a natural number. The choice of $a$ depends on the radix: to represent the quotient using radix $r$, $a$ must be in the range $0, \ldots, r-1$, and there must be at least $r$ values in the range $-a, \ldots, 0, \ldots, a$. The second requirement gives a constraint of $a \geq (r-1)/2$.

Roughly speaking, the algorithm chooses quotient digits as follows: For some values of the partial remainder and divisor, there is more than one quotient digit that will keep the next partial remainder in the desired range. Thus the quotient digit can be selected using an estimate of the partial remainder. For some combinations of $p_j$, $q_j$, and *Divisor*, the next partial remainder $p_{j+1}$ will be negative. When the partial remainder $p_j$ is negative, the algorithm chooses a *negative quotient digit* $q_j$. In the recurrence (1), note that choosing a negative value for a quotient digit results in subtracting a negative value from the partial remainder in the computation $p_j - q_j \cdot Divisor$. In this way, the algorithm tries to choose the quotient digits so that the partial remainders stay in the symmetric range.

The range of the partial remainders $p_j$ is $-k \cdot Divisor \leq p_j \leq k \cdot Divisor$, where $k$ is a constant that depends on the radix $r$ and quotient digit bound $a$. The constant $k$ must be chosen so that the algorithm can maintain $|p_j| \leq k \cdot Divisor$ as an invariant. The following calculation [3] suggests that it is possible to maintain the partial remainders in the desired range if we assign $k = (a \cdot r)/(r-1)$. Suppose $p_j$ is at its maximum value, $p_j = k \cdot Divisor$. In this sitution, the algorithm chooses the largest available quotient digit, $q_j = a$. In order for $p_{j+1}$ to satisfy the upper bound of the range, we must have

$$r \cdot (p_j - q_j \cdot Divisor) = r \cdot (k \cdot Divisor - a \cdot Divisor) \leq k \cdot Divisor.$$

Simplifying, this gives us $r \cdot k - r \cdot a \leq k$, or $k \leq a \cdot r/(r-1)$. The same constraint results from the case $p_j = -k \cdot Divisor$. Thus it seems plausible that a division algorithm can maintain $|p_j| \leq k \cdot Divisor$ as an invariant. In the formal verification in Section 4.2, we show that a particular SRT division circuit with $r = 4$, $a = 2$, and $k = a \cdot r/(r-1) = 8/3$, maintains the invariant $|p_j| \leq (8/3) \cdot Divisor$.

The usual notation for a negative quotient digit is a digit with an overbar; for example, $\bar{2}$ has the value $-2$. A number containing negative digits can be converted to one without negative digits by subtracting the negative digits. As an example, in radix 4, $.21\bar{1} = .203$. The negative digit can be removed by the subtraction $.210 - .001$. In an implementation of the SRT algorithm, it is straightforward to provide hardware that performs the conversion.

For radix 4 calculations, division can be defined using quotient digits $\bar{3}$, $\bar{2}$, $\bar{1}$, 0, 1, 2, 3. Observe, however, that all radix 4 numbers can be represented using only $\bar{2}$, $\bar{1}$, 0, 1, 2. For instance, $.3 = 1.\bar{1}$. This observation allows hardware implementations to avoid the problem of multiplying the divisor by three. We discuss this in more detail in the next section.

*Example 2.* We illustrate SRT division. For radix 10, the quotient digits $\bar{5}, \ldots, 5$ are sufficient to represent all values, so we will choose $a = 5$, $r = 10$. At each step of the computation, we must satisfy the constraint $|p_j| \le k \cdot \mathit{Divisor}$, where $k = a \cdot r/(r-1) = 50/9$. Since the algorithm sets $p_{j+1} = r \cdot (p_j - q_j \cdot \mathit{Divisor})$, we must choose quotient digits at each step to satisfy

$$|p_j - q_j \cdot \mathit{Divisor}| \le \frac{k}{r} \cdot \mathit{Divisor}.$$

Given our choices of $a, r$, we have $k/r = 5/9$. We will compute 1/8.

$$\mathit{Dividend} = 1.000$$
$$\mathit{Divisor} = 8.000$$

*Step 1.* Choose $q_0$ so that $|p_0 - q_0 \cdot \mathit{Divisor}\,| \le \frac{5}{9} \cdot 8.000$. Substituting 1.000 for $p_0$ and 8.000 for *Divisor*, this is $|1.000 - q_0 \cdot 8.000| \le \frac{5}{9} \cdot 8.000$. The only possible choice for the quotient digit $q_0 = 0$. Compute $p_1$:

$$\begin{aligned} p_1 &= 10 \cdot (1.000 - 0 \cdot 8.000) \\ &= 10 \cdot (1.000 - 0) \\ &= 10 \cdot 1.000 \\ &= 10.000 \end{aligned}$$

*Step 2.* Choose $q_1$ so that $|10.000 - q_1 \cdot 8.000| \le \frac{5}{9} \cdot 8.000$. The only possible choice is $q_1 = 1$.

$$\begin{aligned} p_2 &= 10 \cdot (10.000 - 1 \cdot 8.000) \\ &= 10 \cdot (10.000 - 8.000) \\ &= 10 \cdot 2.000 \\ &= 20.000 \end{aligned}$$

*Step 3.* Choose $q_2$ so that $|20.000 - q_2 \cdot 8.000| \le \frac{5}{9} \cdot 8.000$. The constraint is satisfied by choosing $q_2$ to be either 2 or 3. Let us assume that the algorithm chooses $q_2 = 3$.

$$\begin{aligned} p_3 &= 10 \cdot (20.000 - 3 \cdot 8.000) \\ &= 10 \cdot (20.000 - 24.000) \\ &= 10 \cdot (-4.000) \\ &= -40.000 \end{aligned}$$

*Step 4.* Choose $q_3$ so that $|-40.000 - q_3 \cdot 8.000| \leq \frac{5}{9} \cdot 8.000$. The algorithm must select $q_3 = \bar{5}$.

$$\begin{aligned} p_4 &= 10 \cdot (-40.000 - (-5 \cdot 8.000)) \\ &= 10 \cdot (-40.000 - (-40.000)) \\ &= 10 \cdot 0.000 \\ &= 0.000 \end{aligned}$$

The final quotient is $0.13\bar{5} = 0.130 - 0.005 = 0.125$, and the remainder is $p_4 = 0.000$.

## 2.4. *Structure and operation of the division circuit*

The circuit shown in figure 1 is due to Taylor [22]. There are four full-width registers: The Divisor register holds the value of the divisor, the Remainder register holds the value of the partial remainder, and the registers QPOS and QNEG hold the value of the quotient. The q register holds one digit of the quotient. The outputs of the q register are qdigit (2 bits), for the absolute value of the quotient digit, and qsign (1 bit), for the sign. The DALU is a full width adder/subtracter, which is used to compute the partial remainders. The GALU is an 8-bit wide adder/subtracter, which computes an estimate of the partial remainder. QUO LOGIC is a block of combinational logic. Given the leading bits of the divisor and the estimate of the partial remainder from the GALU, QUO LOGIC outputs the next digit of the quotient. At several places, the circuit shifts a signal by one or two bits to the left in order to multiply it by two or four. This operation is shown in the diagram as a box with the operation X 2 or X 4. Throughout the paper, we use roman typeface for names of signals and *italics* for the values of signals.

The full-width registers and the DALU for the circuit in figure 1 can have any width greater than or equal to 10 bits. The reason for this is that the quotient selection logic does not examine any bit beyond the 10 most significant bits of the signals. We show that on each cycle, the circuit correctly computes the basic recurrence for SRT division independent of the word length. Since each cycle produces 2 bits of the quotient, $b$ cycles are required to produce a quotient of length $2b$. The exact sense in which the circuit is independent of the word length will be discussed in Section 4, where we verify a mathematical model of the circuit.

The division circuit operates in two phases: an initialization phase followed by the main calculation phase. The initialization phase begins by setting the Remainder register to hold the dividend, setting the Divisor register to hold the divisor, and setting the QPOS and QNEG registers to zero. After these initializations have been done, the initialization phase uses the GALU and the quotient selection logic to compute the first quotient digit and store it in the q register. This completes the initialization phase.

The calculation phase performs a cycle of the division circuit for each digit beyond the first one. Let us say that cycle $j$ of the circuit is the one in which the value $p_j$ is

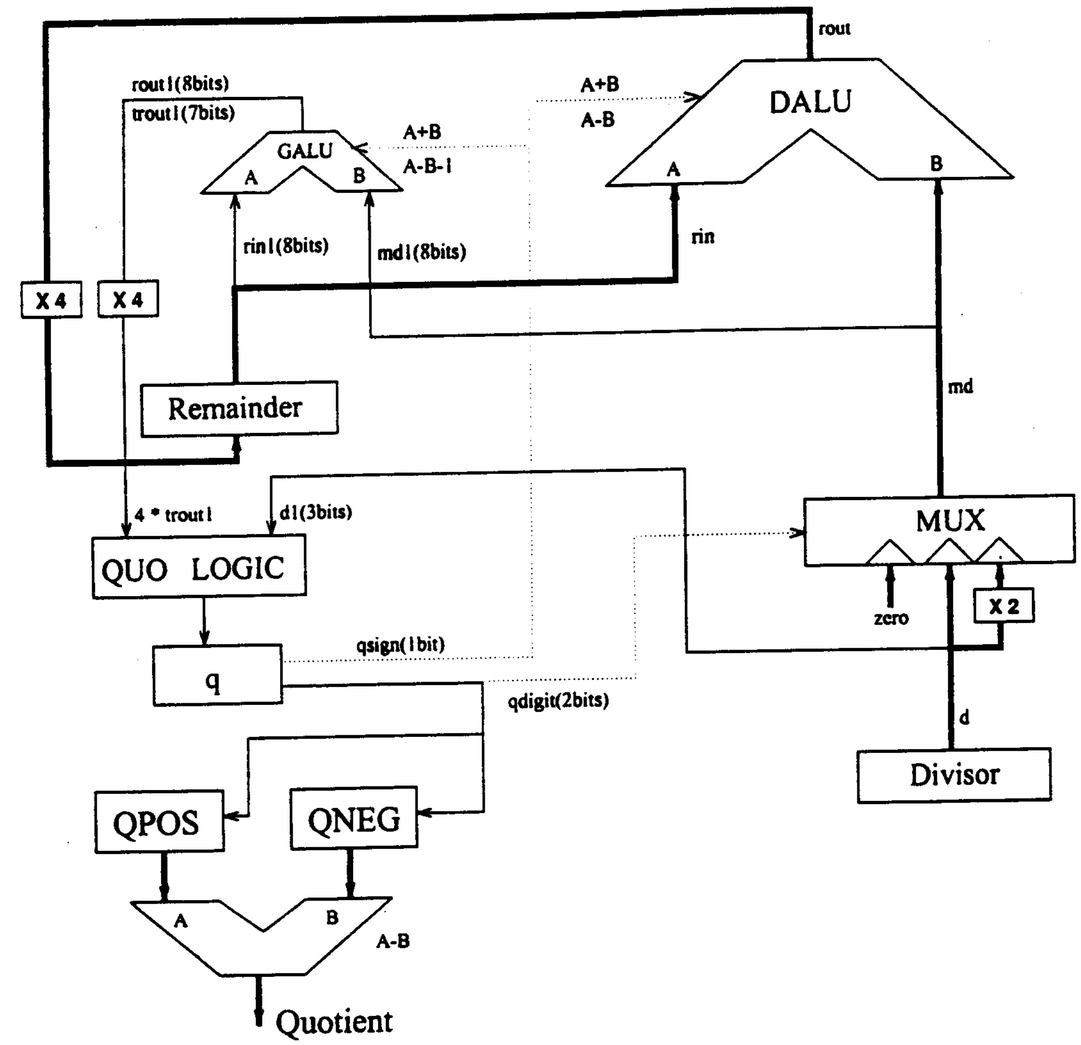

*Figure 1.* The division circuit.

computed. At the beginning of cycle $j + 1$, Remainder holds $p_j$, Divisor holds the divisor, and the q register holds $q_j$. The DALU receives $p_j$ on its A input. The other input to DALU is the signal md, which is controlled by the MUX. The inputs of the MUX are the values 0, *Divisor*, and $2 \cdot$ *Divisor*. Under control of qdigit, the MUX sets the line md to *qdigit* $\cdot$ *Divisor*. The signal qsign controls whether DALU adds or subtracts its inputs: DALU performs subtraction if qsign is +; otherwise it does an addition. The result is that DALU computes the value $p_j - q_j \cdot$ *Divisor* and outputs this value on rout. The signal rout is shifted two bits to the left and stored in the Remainder register for the next cycle. Thus cycle $j + 1$ sets Remainder to the value $p_{j+1}$ in the recurrence (1).

The GALU essentially computes the leading 8 bits of rout. The A (resp. B) input to GALU receives the leading 8 bits of the A (resp. B) input to DALU, and qsign switches GALU between addition and subtraction. The output of GALU is routed through QUO LOGIC to select the next quotient digit. In figure 1, note that the function computed by the GALU is asymmetric: in the addition case, the GALU adds the inputs to compute A + B, while in the subtraction case, the GALU computes A − B − 1. Taylor [22] says that introducing the −1 term produces a better estimate of the partial remainder than having the GALU compute A − B in the subtraction case. The accuracy of the estimate produced by the GALU is crucial for the correctness of the digit selection logic. Since it is not obvious that the design of the GALU combined with the quotient selection logic produces a correct quotient digit in all cases, formal verification of this aspect of the design is of great value.

The value of the quotient is computed using the registers QPOS and QNEG. QPOS holds all of the positive quotient digits and QNEG holds all of the negative digits. On each cycle, these registers are updated as follows: Both registers are shifted two bits to the left. If the digit in the q register is positive, then the value of qdigit (2 bits) is stored in the low-order bits of QPOS and the two low-order bits of QNEG are set to zero. If the digit is negative, then the value of qdigit (i.e., the absolute value of the digit) is stored in the low order bits of QNEG and the low-order bits of QPOS are set to zero. When all of the quotient digits have been computed, the values of QPOS and QNEG are routed to an ALU to compute $QPOS - QNEG$. The output of this operation is the quotient. The reason for storing the positive and negative digits in separate registers is to keep the cycle time of the circuit short. Adding a full-width ALU on the inner cycle of the circuit would slow it down.

### *2.4.1. The quotient selection table.*

The quotient selection logic for QUO LOGIC is represented in tabular form in Table 1. QUO LOGIC receives two inputs: an estimate of the partial remainder from GALU and the first four bits of the divisor, and selects one of the digits $\bar{2}$, $\bar{1}$, 0, 1, 2. In the circuit diagram, the signal trout1 is the result of truncating rout1 to 7 bits, by dropping the low-order bit. The value that is supplied to the quotient lookup table is trout1 multiplied by 4. In the table, the GALU input is $g_7g_6g_5g_4 \cdot g_3g_2g_1$; note $g_7$ is the most significant bit. The table does not list the input values for the least significant bits $g_2g_1$. The reason is that for most values of the inputs, the quotient digit can be determined using only the five leading bits of the GALU output. The bits $g_2g_1$ are needed only near boundaries where the value of the quotient digit changes. The output in these cases is given by the lettered formulas A, B, C, D, E. For example, the formula A says that the quotient digit is −2 unless both $g_1$ and $g_2$ have the value 1, in which case the quotient digit is −1. The other formulas can be read in a similar way.

For input combinations that cannot be reached on executions of the division circuit, the table has no entry, indicated by –. It is important to verify both that the computation stays within the marked area in the table, and that the quotient selections in this part are correct.

## 3. Analytica

In this section, we describe a new approach to mechanical theorem proving that involves combining an automatic theorem prover with a symbolic computation system. The theorem prover, which we call *Analytica*, is able to exploit the mathematical knowledge that is built into this symbolic computation system. In addition, it can guarantee the correctness of

*Table 1.* The quotient prediction table for the division circuit.

| | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | (4 * routl -- 7 bits) | | | | | | | | | | | | | | | | | | | | | |
| g7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| g6 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| g5 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| g4 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| g3 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| g2 | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| |
| g1 | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| | \| |
| 1.000 | -- | -- | -- | -- | -2 | -2 | -2 | A | -1 | -1 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | -- | -- | -- | -- | -- |
| 1.001 | -- | -- | -- | -- | -2 | -2 | -2 | B | -1 | -1 | 0 | 0 | 1 | 1 | C | 2 | 2 | 2 | -- | -- | -- | -- |
| 1.010 | -- | -- | -- | -2 | -2 | -2 | -2 | -1 | -1 | D | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | -- | -- | -- |
| 1.011 | -- | -- | -2 | -2 | -2 | -2 | B | -1 | -1 | D | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | -- | -- | -- |
| 1.100 | -- | -- | -2 | -2 | -2 | -2 | -1 | -1 | -1 | 0 | 0 | 0 | E | 1 | 1 | C | 2 | 2 | 2 | 2 | -- | -- |
| 1.101 | -- | -2 | -2 | -2 | -2 | -2 | -1 | -1 | -1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | -- |
| 1.110 | -2 | -2 | -2 | -2 | -2 | B | -1 | -1 | -1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | -- |
| 1.111 | -2 | -2 | -2 | -2 | -2 | -1 | -1 | -1 | -1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |

(dl -- 4 bits)

A = -(2 - g2 * g1)
B = -(2 - g2)
C = 1 + g2
D = -(1 - g2)
E = g2

certain steps that are made by the symbolic computation system and, therefore, prevent common errors like division by an expression that may be zero.

*Analytica* is written in the Mathematica programming language and runs in the interactive environment provided by this system [24]. Since we wanted to generate proofs that were similar to proofs constructed by humans, we have used a variant of the sequent calculus in the inference phase of our theorem prover. However, quantifiers are handled by skolemization instead of explicit quantifier introduction and elimination rules. Although inequalities play a key role in all of analysis, Mathematica is only able to handle very simple inequalities. We have implemented the Sup-Inf method of Bledsoe [5] to handle linear inequality systems. In addition, we have developed a technique that is able to handle a large class of nonlinear inequalities as well. This technique is more closely related to the BOUNDER system developed at MIT [19] than to the traditional Sup-Inf method.

Analytica consists of four different phases: skolemization, simplification, inference, and rewriting. When a new formula is submitted to Analytica for proof, it is first skolemized to a quantifier free form. Then, in the simplification phase, a large number of rules are used to simplify the atomic formulas (i.e., equations and inequalities) with respect to the current *proof context*. If the formula reduces to true, the current branch of the inference tree terminates with success. If not, the theorem prover matches the formula against the conclusions of the available inference rules, and attempts to prove the formula by backwards chaining.

If Analytica is attempting to prove a goal and no inference rule is applicable, then Analytica tries to use rewriting to convert the goal into another equivalent form. If the formula can be rewritten, then the simplification, inference, and rewriting phases are applied to the new formula. Backtracking will cause the entire inference tree to be searched before the proof of the original goal formula terminates with failure.

- *Skolemization phase*. In Analytica (as in Bledsoe's *UT Prover* [4]), we use skolemization to deal with the quantifiers that occur in the formula to be proved. Initially, quantified variables are standardized so that each has a unique name. After replacing the quantified variables by *Skolem functions* or *Skolem variables*, we can obtain a quantifier-free formula. A formula is valid if and only if it is valid after skolemization. Although formulas are represented internally in skolemized form without quantifiers, quantifiers are added when a formula is displayed so that proofs will be easier to read.
- *Simplification phase*. Simplification is the key phase of Analytica. A formula is simplified with respect to its *proof context*. Intuitively, the proof context consists of the formulas that may be assumed true when the formula is encountered in the proof. The formula that results from simplifying $f$ under context $C$ is denoted by $\mathit{simplify}(f, C)$. In order for the simplification procedure to be sound, $\mathit{simplify}(f, C)$ must always satisfy the following condition:

$$C \models \mathit{simplify}(f, C) \leftrightarrow f.$$

  The initial context $C_0$ in each simplification phase is a conjunction of all of the *given* properties of the variables and constants in the theorem. The initial formula in each simplification phase is the current goal of the theorem prover. In the first simplification phase it is the result of the skolemization phase. In each subsequent simplification phase it is the result of the previous rewriting phase. A large number of rules are provided for simplifying atomic formulas (i.e., equations and inequalities) using context information.
- *Inference phase*. The inference phase is based on the *sequent calculus* [11]. We selected this approach because we wanted our proofs to be readable. Suppose that $f$ is the formula that we want to prove. In this phase we attempt to find an instantiation for the skolem variables that makes $f$ a valid ground formula. In order to accomplish this, $f$ is decomposed into a set of *sequents* using rules of the sequent calculus. Each sequent has the form $\Gamma \vdash \Delta$, where $\Gamma$ and $\Delta$ are initially sets of subformulas of $f$. The formula $f$ will be proved if a substitution can be found that makes all of the sequents valid. A sequent $\Gamma \vdash \Delta$ is valid if it is impossible to make all of the elements of $\Gamma$ true and all of the elements of $\Delta$ false.
- *Rewriting Phase*. Five rewriting tactics are used in Analytica:

  1. Replace the left hand side of an equation in the hypothesis by its right hand side.
  2. Rewrite a trigonometric expression to an equivalent form.

3. Move all terms in equations or inequalities to left hand side and factor the expression.
4. Solve linear equations.
5. Replace a user defined function by its definition.

Special tactics are included in the inference phase for handling inequalities. Inequalities play a key role in all areas of analysis. Although many inequality formulas can be handled in the simplification phase, some valid inequality formulas cannot be reduced to true in this phase. For example, $(x \leq 0 \land y \leq x) \to y \leq 0$ cannot be proved by the technique used in the simplification phase alone. Other more powerful techniques for deciding satisfiability of inequality formulas must be used in addition. If an inequality $a \leq b$ is not directly provable using the techniques in the simplification phase, then Analytica will try to find a term $c$, such that $a \leq c$ and $c \leq b$ are both provable in the current context. In order to find such a term $c$, we compute a set of upper bounds for $a$ and a set of lower bounds for $b$ by using information provided by the current context. The sets computed are denoted by *Upper*$(a)$ and *Lower*$(b)$, respectively. A term $x$ will be in *Upper*$(a)$ only if $a \leq x$ is true in the current context. Likewise, $x$ will be in *Lower*$(b)$ only if $x \leq b$ is true in the current context. To prove $a \leq b$, it is sufficient to prove that there is some $c \in$ *Upper*$(a)$ such that $c \leq b$ is true or that there is some $c \in$ *Lower*$(b)$ such that $a \leq c$ is true.

There are three main ways of obtaining upper and lower bounds for expressions.

1. Obtain bounds from context information: Upper and lower bounds for an expression are calculated in the current context. For example, when proving $(a \leq b) \lor c$, the upper bounds of $a$ and the lower bounds of $b$ are calculated under the context of $\neg c$.
2. Obtain bounds from the monotonicity of some function: If $f$ is a monotonically increasing function, and $a'$ is an upper (lower) bound of $a$, $f(a')$ is an upper (lower) bound of $f(a)$; if $f$ is a monotonically decreasing function and $a'$ is an upper (lower) bound of $a$, $f(a')$ is a lower (upper) bound of $f(a)$.
3. Use some known bound on the value of a function: If $f$ is bounded, i.e., for all $x$, $f(x) \leq M$, or $f(x) \geq M'$, $M$ is an upper bound for $f(x)$ and $M'$ a lower bound for $f(x)$.

The above technique is complete for linear inequalities, and it can also be used to prove many of the nonlinear inequalities that arise in practice. However, the overhead required for nonlinear inequalities makes the algorithm very inefficient for linear inequalities. Consequently, we have incorporated Bledsoe's Sup-Inf method [5, 21] into Analytica for handling linear inequalities. The Sup-Inf method is treated as a special tactic in the inference phase and is applied before the more complicated inequality reasoning tactic. The Sup-Inf method provides a decision procedure for universally quantified formulas containing linear inequalities. Although this method was initially used for formulas of Presburger arithmetic, it is applicable to the reals as well.

When we apply the Sup-Inf method to a sequent, we first negate the sequent and obtain a conjunction of formulas. We drop all of the conjuncts that are neither equations nor inequalities. If we replace $a = b$ by $a \leq b \land b \leq a$, we can obtain a conjunction of inequalities. The sequent is valid if this conjunction is not satisfiable. When applying this method to complex formulas, we treat nonlinear inequalities as linear ones by replacing each nonlinear term by a new variable. Given a linear inequality system $S$ and a variable $v$, $\mathrm{SUP}_S(v)$ computes the maximum value $v$ can take in real solution of $S$ and $\mathrm{INF}_S(v)$ computes the minimal value. For example, if $S = \{0 \le y,\ x \le y,\ x \le 1 - y\}$, then $\mathrm{SUP}_S(x) = \frac{1}{2}$, $\mathrm{INF}_S(x) = -\infty$, $\mathrm{SUP}_S(y) = \infty$, $\mathrm{INF}_S(y) = 0$. The definition of these functions can be found in [21]. The linear inequality system is not satisfiable if for some variable $v$, $\mathrm{SUP}_S(v) < \mathrm{INF}_S(v)$.

Analytica contains special rules for reasoning about strict inequalities between expressions whose values are integers or integer multiples of a common factor. For example, if $x, y$ are integer-valued, then the inequality $x < y$ is presented to the Sup-Inf algorithm as $x + 1 \le y$. More generally, if $x, y$ are known to be integer multiples of a rational number $p$, where $p > 0$, then $x < y$ is converted to $x + p \le y$. We say that a variable $x$ *has precision* $p$ if the value of $x$ is constrained to be an integer multiple of $p$.

In order to specify the possible values of variables, Analytica programs can contain declarations of the form

```
DeclarePrecision[x] := p;
```

where $x$ is a variable and $p$ is a positive rational number. When this declaration is processed, Analytica records an assumption that the value of $x$ is an integer multiple of $p$.

The following rules are used to determine that an expression is a multiple of a rational number:

1. If $a$ is a constant, then $a$ is a multiple of $\mathrm{Abs}(a)$.
2. If the variable $x$ has been declared with DeclarePrecision[x] := p, then $x$ is a multiple of $p$.
3. If $x$ is a multiple of $p$ and $y$ is a multiple of $q$, for $p, q > 0$, then $x + y$ is a multiple of $\mathrm{GCD}(p, q)$ and $xy$ is a multiple of $pq$. The validity of these rules follows from simple arithmetic.

## 4. Proof of the correctness of the SRT algorithm

In this section, we verify a mathematical model of the circuit. First, we construct the model systematically from the circuit diagram in figure 1. Then we discuss the specifications of the circuit. Analytica proves the correctness of the SRT circuit automatically from the mathematical model and the specifications.

The circuit contains the following operations on signals:

1. Addition and subtraction of scaled integer values.
2. Left shifting of signals to multiply by a power of 2.
3. Table lookup and selection of signals by a multiplexor.
4. Truncation of signals to drop the low-order bits.

We model these operations as follows:

1. Addition and subtraction of signals that represent scaled integer values are modelled as addition and subtraction on the rationals. Initially we model the circuit using unbounded arithmetic. This model corresponds to bounded arithmetic in an actual circuit, provided the circuit does not have arithmetic overflow. We use the unbounded arithmetic model to show that the signal widths of the actual circuit in figure 1 are sufficient to perform all arithmetic operations without overflow.
2. Left shifting of signals is modelled as an operation that multiplies the value of a signal by a constant power of 2.
3. Table lookup and multiplexing are modelled by conditional expressions in the logic of the theorem prover Analytica.
4. Truncation of the low-order bits of a signal is modelled by *truncation inequalities*, which are explained in the next section.

Because the mathematical model of the circuit is defined using a small number of rules based on the circuit structure, it is possible in principle to extract the model mechanically from a description of the circuit structure. In order to do this, we would annotate the circuit with information about the numerical interpretations of the signals. For example, the Remainder signal would be annotated with the information that it is a two's complement number having a certain number of bits on each side of the binary point. Given this information, it is possible to generate the model for Analytica mechanically. One advantage of automatic extraction is that it would be a step towards carrying out design and verification of arithmetic circuits at the same time.

## 4.1. *Axioms for the circuit*

In our model of the circuit, we represent registers by equational rewrite rules that specify the value of a register in the next cycle as a function of the values of signals in the current cycle. An equation of the form

$$next[reg] = expr$$

means that the next value of *reg* is given by the expression *expr*. The notation *next*[*reg*] designates the next value of *reg*. Note that *next*[·] is a special constructor in our equations, not a function symbol.

We need some definitions for two's complement arithmetic. A *binary digit* is 0 or 1. A *binary numeral* is a string of the form $l \bigtriangleup r$ containing at least one binary digit, where $l$ and $r$ are strings of zero or more binary digits and the symbol $\bigtriangleup$ represents the binary radix point. A binary numeral $l \bigtriangleup r$ is said to be in *format* $(nl, nr)$ if the length of $l$ is $nl$ and the length of $r$ is $nr$. A binary numeral $l \bigtriangleup r$ is said to have $nl$ bits left of the binary point if the length of $l$ is $nl$. Consider the binary numeral $N = b_1, \ldots, b_{nl} \bigtriangleup b_{nl+1}, \ldots, b_{nl+nr}$, where for $1 \leq i \leq nl + nr$, $b_i$ is a binary digit, $nl \geq 1$, and the binary point is to the right of $b_{nl}$. The *value of N in two's complement notation* is given by the function

$$Val(N) = -2^{nl-1} \cdot b_1 + \sum_{i=2}^{nl+nr} 2^{nl-i} \cdot b_i.$$

The power of 2 factor associated with each binary digit in the above formula is said to be the *weighting* of that binary digit. Note that the smallest value represented by a two's complement numeral in format $(nl, nr)$ is $-2^{nl-1}$, and the largest value represented is $2^{nl-1} - 2^{-nr}$.

In several places, we need to reason about signals that are formed by dropping the low-order bits from a signal that represents a two's complement numeral. For example, the signal rin1 in figure 1 sends the leading bits of the remainder to the GALU. The full value of the remainder is carried on the signal rin. The signal rin1 contains the leading bits of rin, including six bits to the right of the binary point. Low-order bits beyond this are truncated or effectively set to zero. In the circuit, the remainder and the signal md are represented using two's complement arithmetic. The following diagram shows the arithmetic weightings of the bits in the signals rin and rin1 in two's complement notation. The numeric value of a two's complement signal is the sum of the weightings of the bits that are on. The diagram assumes that rin has $n$ bits, for $n \geq 10$; the most significant bit is $b_1$.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *bit* : | $b_1$ | $b_2$ | $b_3$ | $b_4$ | $\Delta$ | $b_5$ | $b_6$ | $b_7$ | $b_8$ | $b_9$ | $b_{10}$ | $b_{11}$ | $\cdots$ | $b_k$ | $\cdots$ | $b_n$ |
| rin : | $-2^3$ | $2^2$ | $2^1$ | $2^0$ | $\Delta$ | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | $2^{-5}$ | $2^{-6}$ | $2^{-7}$ | $\cdots$ | $2^{3-k}$ | $\cdots$ | $2^{3-n}$ |
| rin1 : | $-2^3$ | $2^2$ | $2^1$ | $2^0$ | $\Delta$ | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | $2^{-5}$ | $2^{-6}$ | | | | | |

Since the bits of rin that are not included in rin1 have positive weightings, $rin1 \leq rin$. Also, we have

$$rin - rin1 \leq \sum_{i=-7}^{3-n} 2^i < 2^{-6}.$$

This gives us the following inequality:

$$rin1 \leq rin < rin1 + 2^{-6} \tag{4}$$

It is important that this inequality depends only on the bit position relative to the binary point at which truncation of the full signal occurs. In particular, the inequality does not depend on the number of bits that the full-width signal has on either side of the binary point. This means the inequality is valid in both the bounded arithmetic of an actual circuit and the unbounded arithmetic model that we use for representing addition and subtraction.

Interestingly, the inequality in (4) is sufficient for our proof, even though it does not capture all of the mathematical properties of truncation. The exact result of truncating a signal such as rin is a staircase function, which remains level for each interval in which $b_1 \cdots b_{10}$ are fixed. This staircase function does not have as small a representation as (4) using linear inequalities. In the proof, we use inequalities like (4) several times to reason about signals that are truncated.[1]

We are now ready to write down the axioms for the signals of the division circuit. In the equations, the variable qdigit represents the absolute value of the quotient digit, and the variable qsign is a boolean representing the sign (*qsign* is true iff the sign is $+1$).

- The MUX:

$$md = \begin{cases} 0 & \text{when } qdigit = 0 \\ d & \text{when } qdigit = 1 \\ 2d & \text{when } qdigit = 2 \end{cases}$$

- The DALU:

$$rout = \begin{cases} rin - md & \text{when } qsign \\ rin + md & \text{when } \neg qsign \end{cases}$$

- The Remainder:

$$next[rin] = 4 \cdot rout$$

- The QPOS:

$$next[QPOS] = \begin{cases} 4 \cdot QPOS + qdigit & \text{when } qsign \\ 4 \cdot QPOS & \text{when } \neg qsign \end{cases}$$

- The QNEG:

$$next[QNEG] = \begin{cases} 4 \cdot QNEG & \text{when } qsign \\ 4 \cdot QNEG + qdigit & \text{when } \neg qsign \end{cases}$$

- The Quotient:

$$Quotient = QPOS - QNEG$$

- The signal rin1 is rin truncated to 6 bits after the binary point:

$$rin1 \leq rin < rin1 + 2^{-6}$$

- The signal md1 is md truncated to 6 bits after the binary point:

$$md1 \leq md < md1 + 2^{-6}$$

- The GALU:

$$routl = \begin{cases} rinl - mdl - 2^{-6} & \text{when } qsign \\ rinl + mdl & \text{when } \neg qsign \end{cases}$$

In the subtraction case, the term $-2^{-6}$ reduces the GALU output by one *ulp*. This is intended to improve the accuracy of the GALU's estimate of the partial remainder [22]. We have used Analytica to show formally that the GALU satisfies the following inequality:

$$routl \le rout < routl + 2^{-5} \tag{5}$$

Roughly speaking, the circuit achieves these bounds as follows: Both of the inputs to the GALU are formed by truncating signals that are inputs to the DALU. Each of the truncated signals satisfies an inequality

$$xl \le x < xl + 2^{-6}, \tag{6}$$

where $x$ is the full-width input to DALU and $xl$ is the truncated signal. In the case that the GALU adds its two inputs, this implies that (5) holds, because the GALU adds two signals that satisfy inequalities of the form (6). In the subtraction case, truncation works in the opposite direction for the term $-mdl$, i.e., we have

$$-mdl - 2^{-6} < -md \le -mdl.$$

The GALU computes $rinl - mdl - 2^{-6}$; the term $2^{-6}$ is intended to compensate for the effect of truncation on $-mdl$. However, care is needed to analyze this circuit properly. In the absence of formal verification, it is not obvious that the GALU and the quotient selection table give correct quotient digits. In the next section, we discuss the properties we have formally verified to show that the GALU has enough accuracy to produce correct quotient digits with Taylor's quotient selection table.

- The signal d1 is the leading 4 bits of d (constant 1 before binary point and 3 bits after). The divisor is a normalized number, so the signal d represents an unsigned binary number with the most significant bit set to 1. The signals d and d1 have the following formats:

$$\mathrm{d}\ :\ \boxed{1}\ \triangle\ \boxed{2^{-1}}\boxed{2^{-2}}\boxed{2^{-3}}\boxed{2^{-4}}\cdots$$

$$\mathrm{d1}\ :\ \boxed{1}\ \triangle\ \boxed{2^{-1}}\boxed{2^{-2}}\boxed{2^{-3}}$$

We use the following truncation inequality as an axiom in our model:

$$dl \le d < dl + 2^{-3}$$

Additionally, we make use of the discreteness of d1. Because of its format, d1 can only have the 8 binary values 1.000, 1.001, 1.010, 1.011, 1.100, 1.101, 1.110 and 1.111. This limitation on the range of d1 is expressed in our model by the following formula:

$$dl = 1 \vee dl = \frac{9}{8} \vee dl = \frac{5}{4} \vee dl = \frac{11}{8} \vee dl = \frac{3}{2} \vee dl = \frac{13}{8} \vee dl = \frac{7}{4} \vee dl = \frac{15}{8}$$

- The signal trout1 is the leading 7 bits of rout1 (4 bits before the binary point and 3 bits after). The following axiom says that trout1 is produced by truncating rout1:

$$routl \leq troutl \leq routl + 2^{-6}$$

The proof constructed by Analytica also uses the fact that the value of trout1 is a multiple of $2^{-5}$.

- The QUO LOGIC: The quotient logic receives inputs d1 and trout1, and produces the next quotient digit. This is the hardest part in formalizing the circuit. Intuitively, the quotient logic can be represented as a large conditional expression. In our initial approach [12] using the Maple symbolic algebra system, we formed a separate case for each element of the quotient table. The cases were defined in a bit-level definition of the quotient table. For example, a typical table entry with this approach says that if the four bits of $dl$ = 1.010, and the five bits of the GALU are $troutl$ = 00.101 then $next[qdigit] = 2$.

  The initial approach to modelling the QUO LOGIC resulted in long running times for the proof, because each entry in the quotient table was treated as a separate case. We reduced the number of cases in the proof by representing each row of the quotient prediction table as a *boundary value list* $\{b_1, b_2, b_3, b_4, b_5, b_6\}$. For a given value of $dl$, we choose $b_6$ to be the minimal positive value for $(4 \cdot troutl)$ that is outside the defined quotient values in the table. For example, when $dl = 1$, this minimal value has binary representation 0011.0. Consequently, $b_6 = 3$. Similarly, we choose $b_1, b_2, b_3, b_4$ and $b_5$ to be the minimal values for $(4 \cdot trout1)$ that gives quotient values $-2, -1, 0, 1$ and 2, respectively. When $dl = 1$, the minimal value for $(4 \cdot troutl)$ with quotient $-2$ has binary two's complement representation 1100.1. Therefore, $b_1 = -7/2$. The boundary value list for each of the 8 possible values for $dl$ are shown below:

| | | | | | | |
|---|---|---|---|---|---|---|
| {−7/2, | −13/8, | −1/2, | 1/2, | 3/2, | 3}, | when $dl = 1$; |
| {−7/2, | −7/4, | −1/2, | 1/2, | 7/4, | 7/2}, | when $dl = 9/8$; |
| {−4, | −2, | −3/4, | 1/2, | 2, | 4}, | when $dl = 5/4$; |
| {−9/2, | −9/4, | −3/4, | 1/2, | 2, | 4}, | when $dl = 11/8$; |
| {−9/2, | −5/2, | −1, | 3/4, | 9/4, | 9/2}, | when $dl = 3/2$; |
| {−5, | −5/2, | −1, | 1, | 5/2, | 5}, | when $dl = 13/8$; |
| {−11/2, | −11/4, | −1, | 1, | 5/2, | 5}, | when $dl = 7/4$; |
| {−11/2, | −3, | −1, | 1, | 3, | 11/2}, | when $dl = 15/8$; |

From the definition of the boundary values, we know that the following holds:

1. when $b_1 \le 4 \cdot troutl < b_2$, $q = -2$;
2. when $b_2 \le 4 \cdot troutl < b_3$, $q = -1$;
3. when $b_3 \le 4 \cdot troutl < b_4$, $q = 0$;
4. when $b_4 \le 4 \cdot troutl < b_5$, $q = 1$;
5. when $b_5 \le 4 \cdot troutl < b_6$, $q = 2$;
6. when $4 \cdot troutl < b_1$, out of table and we define $q = -3$;
7. when $4 \cdot troutl \ge b_6$, out of table and we define $q = 3$;

Let $\{b_1, b_2, b_3, b_4, b_5, b_6\}$ represent the row in the quotient prediction table that corresponds to $d1$. The QUO LOGIC is given by:

$$next[qsign] = (4 \cdot troutl \ge b_3)$$

$$next[qdigit] = \begin{cases} 3 & \text{when } 4 \cdot troutl < b_1 \vee 4 \cdot troutl \ge b_6 \\ 2 & \text{when } b_1 \le 4 \cdot troutl < b_2 \vee b_5 \le 4 \cdot troutl < b_6 \\ 1 & \text{when } b_2 \le 4 \cdot troutl < b_3 \vee b_4 \le 4 \cdot troutl < b_5 \\ 0 & \text{when } b_3 \le 4 \cdot troutl < b_4 \end{cases}$$

## 4.2. *Specifications and proof of the circuit*

The main issue we address in verification of the SRT division circuit is the correctness of the quotient lookup table with respect to the surrounding data path. We abstract away standard components such as the ALU. We also abstract the control logic of the circuit; the control logic is essentially a counter that counts up to the number of cycles needed to compute the quotient and then outputs the result. Because this control logic is *data independent*, it would not be difficult to obtain good coverage with testing. In contrast, known methods of testing provide very poor coverage for the quotient lookup table [16].

The correctness of the main calculation phase of the circuit depends on two invariants:

$$next[rin + 4 \cdot Quotient \cdot d] = 4 \cdot (rin + 4 \cdot Quotient \cdot d) \qquad \text{(Inv1)}$$

$$-\frac{2}{3} \cdot d \le rout < \frac{2}{3} \cdot d \qquad \text{(Inv2)}$$

The first invariant says that $(rin + 4 \cdot Quotient \cdot d)$ remains constant with respect to left shifting by 2 bits. The initial value of this expression is *Dividend*. We will use the fact that this expression does not change value to show that the final quotient and remainder are correct.

The second invariant guarantees that the computation will never overflow (cf. [3]). We regard Inv2 as the main property of the circuit to be verified, because it implies that the quotient lookup table is correct. The constant 2/3 in Inv2 is determined by the choice of the radix $r = 4$ and the range of quotient digits, $-2, \ldots, 2$, for this circuit. The general formula to determine the bound on *rout* is part of the basic theory of SRT division (cf. [3]). Since the formula for the bound on rout is well-known to practising circuit designers, the invariant Inv2 is generally known when an SRT divider is being designed. It is also known from the basic theory of SRT division that Inv1 and Inv2 imply that the computed quotient converges to the correct value as a function of the number of cycles the algorithm runs. At the end of this section, we apply the invariants to reproduce this result.

Analytica proves the following invariance properties automatically:

- The assertion Inv1 is an invariant.

$$next[rin + 4 \cdot Quotient \cdot d] = 4 \cdot (rin + 4 \cdot Quotient \cdot d)$$

- The GALU gives an estimate for the remainder that satisfies the following inequality:

$$routl \leq rout < routl + 2^{-5}$$

- The remainder never falls outside of the defined part of the quotient table.

$$-\frac{2}{3} \cdot d \leq rout < \frac{2}{3} \cdot d \Longrightarrow next[qdigit] = 0 \vee next[qdigit] = 1 \vee next[qdigit] = 2$$

- The assertion Inv2 is an invariant. (We assume that Inv2 holds after the circuit initialization phase, and use Analytica to prove by induction that it remains true in the main calculation phase.)

$$-\frac{2}{3} \cdot d \leq rout < \frac{2}{3} \cdot d \Longrightarrow -\frac{2}{3} \cdot d \leq next[rout] < \frac{2}{3} \cdot d$$

All of the theorems are proven by Analytica. The last theorem is the most interesting. The exact statement of the theorem is given below.

```
Prove[imp[and[d1 <= d < d1 + 2^(-3),
               or[d1 == 8/8, d1 == 9/8, d1 == 10/8, d1 == 11/8,
                  d1 == 12/8, d1 == 13/8, d1 == 14/8, d1 == 15/8],
               routl <= rout < routl+2^(-5),
               troutl <= routl <= troutl + 2^(-6),
               -2/3 d <= rout < 2/3 d],
        -2/3 d <= next[rout] < 2/3 d]];
```

Notice that there are some additional conjuncts in the hypothesis part. The first two hypotheses are axioms about the values of d1. The third conjunct, relating rout and routl, states that GALU gives a correct estimate for the remainder. Analytica proves the theorem about the GALU separately, so we can assume it as a hypothesis in this proof. The fourth conjunct says that troutl is the result of truncating the $2^{-6}$ bit of routl. The whole input required by Analytica to prove these theorems and part of the proof it generates are shown in Appendices A.1 and A.2.

As an application of the main invariance results described above, we show how to use Inv1 and Inv2 to prove that the quotient converges to the correct value. This part of the proof can be regarded as part of the known theory of SRT division [3], and less a property specific to the circuit we are verifying. The steps of this proof have been checked by Analytica, with about 10 user-supplied lemmas.

Let $Quotient_0$ (resp. $rin_0$, $rout_0$) be the initial value of Quotient (resp. rin, rout), and let $Quotient_j$ (resp. $rin_j$, $rout_j$) be the values of these variables at the end of the $j$th cycle, for $j = 1, \ldots, l$, where $l \geq 1$ is the number of radix 4 digits in the quotient. The initial conditions are $Quotient_0 = 0$, $rin_0 = Dividend$.

The following formula can be proved by induction on the number of cycles $j$:

$$Dividend = \frac{rin_j + 4 \cdot d \cdot Quotient_j}{4^j} = \frac{rin_j}{4^j} + d \cdot \frac{Quotient_j}{4^{j-1}} \qquad \text{(Prop1)}$$

The base case for $j = 0$ follows from the initialization. The inductive case is just Inv1. Intuitively, this formula says that the remainder is $rin_j$ scaled by $4^{-j}$ and the quotient is $Quotient_j$ scaled by $4^{1-j}$.

It remains to show that the scaled value of the quotient converges to $Dividend/d$. First, we will induct on $j$, to show that $|rout_j| \leq \frac{2}{3} \cdot d$, for $j \geq 0$. The base case follows from the initialization of the circuit, which selects the first quotient digit. The inductive case is Inv2. Since $rin_{j+1} = 4 \cdot rout_j$, for $j \geq 0$, we can infer that $|rin_j| \leq \frac{8}{3} \cdot d$ for $j \geq 1$.

Using Prop1, we have a bound on the difference between the infinitely precise quotient and the computed quotient:

$$\left| \frac{Dividend}{d} - \frac{Quotient_j}{4^{j-1}} \right| = \left| \frac{rin_j}{d \cdot 4^j} \right| \leq \frac{8}{3} \cdot \frac{d}{d \cdot 4^j} = \frac{8}{3 \cdot 4^j} = \frac{2}{3} \cdot 4^{1-j} < 4^{1-j}$$

Thus the error in the quotient at step $j$ is less than $4^{1-j}$.

## 4.3. *Correct implementation on a finite-width data path*

Thus far, we have analyzed the circuit in an unbounded arithmetic model. Now we use the results of this analysis to show that the circuit can be correctly implemented with fixed-width data paths of certain sizes. In this discussion, we focus on the data path that calculates the Remainder. We first discuss the width of the Remainder register the signal md, then we discuss the widths of the arithmetic units.

We have shown that the partial remainders satisfy $|p_j| \leq \frac{8}{3} \cdot d$, where $1 \leq d < 2$. It follows that

$$|p_j| < \frac{16}{3} = 5\frac{1}{3}.$$

The weightings of the bits to the left of the binary point in a format $(4, n)$ numeral are $-8$, 4, 2, 1; thus the remainder can be represented as a $(4, n)$ numeral. On the right of the binary point, at least 6 bits are needed, because we used the property that the GALU input signal rin1 contains 6 bits to the right of the binary point in showing the accuracy of the GALU estimate.

Now consider the signal md, which represents the product of the divisor and the unsigned quotient digit. Since the divisor is in the range $1 \leq Divisor < 2$ and the unsigned quotient digit is 0, 1, or 2, md is in the range $0 \leq md < 4$. This means that md can be represented using three bits to the left of the binary point and a constant fourth sign bit which is always 0 (because md is non-negative).

Now we will consider the widths of the arithmetic units. Because the delay time of an adder is proportional to its bit width, the smallest possible width should be used. On the other hand, if the width is too small, arithmetic overflow will occur, so we need to analyze the widths carefully. In fact, the width of the GALU presents a puzzle: As we have discussed, the Remainder has format $(4, n)$, and we have also seen that the GALU receives 6 input bits to the right of the binary point to achieve the necessary accuracy. From this, it may appear that the GALU needs to be at least 10 bits wide. However, Taylor [22], says that the GALU is implemented as 8-bit ALU but does not give a clear explanation of why 8 bits are sufficient.

We will show that the GALU can be implemented as an 8-bit arithmetic unit. As a preliminary, observe that the output of both the DALU and GALU can be represented with two bits to the left of the binary point. We have seen that the operand inputs to the arithmetic units have four bits to the left of the binary point. In general, if two operands having four bits to the left of the binary point are added or subtracted, the result will occupy five bits to the left of the binary point. However, we have shown that in the division circuit, the correct choice of quotient digits keeps the output of the DALU in the range $|rout| \leq \frac{2}{3} \cdot d < \frac{4}{3}$. This means that rout can be represented using only two bits to the left of the binary point, weighted $-2$, 1. This is consistent with the output of the DALU having format $(2, n)$, for $n \geq 6$.

For the GALU, we have shown that the GALU output rout1 approximates rout in the sense

$$rout1 \leq rout \leq rout + 2^{-5}.$$

This implies that the output of the GALU satisfies

$$-2 \leq rout1 \leq 2 - 2^{-6}, \tag{7}$$

making rout1 representable in format $(2, 6)$.

The output of the DALU is shifted two bits to the right and stored in the Remainder register. Note that when a number in format $(2, n)$ is shifted two bits to the left, it results in a number with four bits to left of the binary point, which is the format we want for the Remainder register.

Now that we have shown the output of the GALU is representable in format $(2, 6)$, we need to show that the correct output can be obtained using an 8-bit ALU. Although the implementation of the GALU is not explained clearly in [22], we show how a correct implementation can be defined. The leading 10 bits of the Remainder and md are numerals in format $(4, 6)$. The GALU drops two leading high-order bits of each signal, and performs arithmetic on the remaining numerals in format $(2, 6)$. The DALU can also drop the two highest order bits of its inputs. We will now show that this arrangement produces the correct results.

The arithmetic operation performed by the GALU is either an addition or a subtraction; in the subtraction case, the GALU actually computes A−B−1. We will consider the addition case first. In order to discuss addition of binary numerals formally, let us define $Sum(x, y)$ to be the function mapping two binary numerals in format $(nl, nr)$ into a numeral in format $(nl + 1, nr)$, with

$$Val(Sum(x, y)) = Val(x) + Val(y).$$

We also need to define the *restriction* of a binary numeral to a smaller format. Let $N$ be a binary numeral in format $(nl, nr)$, with $N = l \Delta r$, where $l = l_{nl}, \dots, l_1, r = r_1, \dots, r_{nr}$. If $nl' \le nl$ and $nr' \le nr$, then we define $N|_{(nl', nr')}$ to be the binary numeral $l_{nl'}, \dots, l_1 \Delta r_1, \dots, l_{nr'}$.

The following theorem says that if $N$ is a binary numeral with at least two bits to the left of the binary point and the value of $N$ is in the range that can be represented using only two bits to the left of the binary point, then a numeral with the same value can be formed by truncating the high-order bits down to two bits. The theorem is easily proved by considering the weightings of the leading bits.

**Theorem 1.** *If $N$ is a binary numeral in format $(nl, 6)$, with $nl \ge 2$ and $-2 \le Val(N) \le 2 - 2^{-6}$, then $Val(N) = Val(N|_{(2,6)})$.*

Applying this result to the circuit, we can infer, for example, that since the output of the GALU is in the range given in (7), a numeral representing rout1 can be correctly computed the following procedure:

1. Use an adder with inputs in format (4, 6) and output in format (5, 6); this adder will not overflow.
2. Truncate the output of the adder to format (2, 6).

The above procedure is redundant in the sense that truncated output bits of the adder do not need to be calculated. This is expressed by the following theorem. The theorem is a consequence of fact that each output bit of the *Sum* function is independent of all input bits that are relatively higher order (i.e., located to the left of the given output bit).

**Theorem 2.** *If $M, N$ are binary numerals in format $(nl, nr)$ for some $nl, nr$ and $k$ is a natural number, $k \le nl$, then*

$$Sum(M, N)|_{(k,nr)} = Sum(M|_{(k,nr)}, N|_{(k,nr)})_{(k,nr)}.$$

We can now conclude that in the addition case, the GALU produces the correct output by truncating the inputs to (2, 6) format and adding the truncated inputs.

For the subtraction case, we use the *one's complement* of the B input. Formally, if $N$ is a binary numeral in format $(nl, nr)$, then define $Comp(N)$ to be the result of replacing each 1 in $N$ with a 0 and vice versa. If $N$ is in format $(nl, nr)$, then $Comp(N)$ is a binary numeral in the same format, and the values are related by

$$Val(Comp(N)) = -Val(N) - 2^{-nr}.$$

The subtraction case of the GALU on inputs $M, N$ can be implemented as follows:

1. Truncate both inputs to format (2, 6).
2. Compute $Sum(M, Comp(N))$.

By our argument in the addition case, this procedure produces the correct result for the GALU.

While the above proof is conceptually simple, applying manual arguments to circuit designs is a notably error-prone process. It is therefore of interest to mechanize the reasoning. We will now discuss one approach that can be used to mechanize the correctness proof for the GALU in Analytica.

First, we use Analytica to verify that the bits of DALU output weighted $-16, 8, 4$ do not need to be computed. For the DALU and the output signal rout, we formulate the problem using binary variables, b1, ..., b5, to represent the higher order bits of a two's complement number. The low-order bits are represented by a rational variable frac. All of the Analytica theorems shown in this section are proven automatically.

```
Prove[imp[and[-4/3 <= rout <= 4/3,
              rout == -16*b1 + 8*b2 + 4*b3 + 2*b4 + b5 + frac,
              b1 == 1 || b1 == 0,
              b2 == 1 || b2 == 0,
              b3 == 1 || b3 == 0,
              b4 == 1 || b4 == 0,
              b5 == 1 || b5 == 0,
              0 <= frac < 1],
          and[b1 == b4, b2 == b4, b3 == b4,
              rout == -2*b4 + b5 + frac]]];
```

The theorem says that if *rout* is in the range $|rout| \leq 4/3$ and it is represented as a two's complement number with five bits to the left of the binary point, then the four highest order bits must have the same binary value, and rout can also be represented as a two's complement number using only b4 and b5 to the left of the binary point.

For the GALU and output rout1, we prove a similar result, using the previously proved result that rout1 approximates rout to within $2^{-5}$.

```
Prove[imp[and[-4/3 <= rout <= 4/3,
              approx[rout1, rout, 5],
              rout1 == -16*b1 + 8*b2 + 4*b3 + 2*b4 + b5 + frac,
              b1 == 1 || b1 == 0,
              b2 == 1 || b2 == 0,
              b3 == 1 || b3 == 0,
              b4 == 1 || b4 == 0,
              b5 == 1 || b5 == 0,
              0 <= frac < 1],
          and[b1 == b4, b2 == b4, b3 == b4,
              rout1 == -2*b4 + b5 + frac]]];
```

Now we consider the arithmetic operations on the truncated inputs. We first prove two preliminary results showing that if the truncated inputs are interpreted as ***unsigned*** binary numerals, then the results of performing arithmetic on the truncated inputs fall in certain ranges. The unsigned interpretation is used because when the inputs are truncated, the leading (negative weight) bit is dropped. In these theorems, we represent the truncated inputs as the sum of a bit with weight 2 and a rational number, lowx or lowy, which represents the value of the input bits contained in format $(1, 6)$. Two theorems are proved

```
(* The following theorem implies that truncx + truncy is
   representable as 4*z1 + 2*z2 + lowz. *)

Prove[imp[and[k == 2^(-6),
              or[x3 == 1, x3 == 0],
              or[y3 == 1, y3 == 0],
              0 <= lowx <= 2-k,
              0 <= lowy <= 2-k,
              truncx == 2*x3 + lowx,
              truncy == 2*y3 + lowy],
          0 <= truncx + truncy <= 8-k]];
```

for the GALU, one for the addition case and one for the subtraction case. We interpret the results to mean that the arithmetic value produced by the GALU can be represented in different ways: For the addition case, the arithmetic value can be represented as an unsigned binary numeral in format $(3, 6)$. For the subtraction case, the value can be represented as a two's complement binary numeral in format $(3, 6)$.

Finally, we prove a main result for each of the two cases of the GALU. In these theorems, the variables xin and yin stand for the two inputs to the GALU in format $(4, 6)$, before truncation of the two high-order bits. The variables truncx and truncy stand for the result of truncating the two inputs to format $(2, 6)$. For the addition case, the theorem says that if $truncx + truncy$ is representable as $4 \cdot z1 + 2 \cdot z2 + lowz$, for $0 \leq lowz < 2$, then the sum of the untruncated inputs xin and yin is equal to the sum of the truncated inputs. For the subtraction case, the theorem says that if *truncx* $-$ *truncy* $- 2^{-6}$ is representable as

```
(* The following theorem implies that truncx - truncy - k is
   representable as -4*z1 + 2*z2 + lowz. *)

Prove[imp[and[k == 2^(-6),
              or[x3 == 1, x3 == 0],
              or[y3 == 1, y3 == 0],
              0 <= lowx <= 2-k,
              0 <= lowy <= 2-k,
              truncx == 2*x3 + lowx,
              truncy == 2*y3 + lowy],
           -4 <= truncx - truncy - k <= 4-k]];
```

$-4 \cdot z1 + 2 \cdot z2 + lowz$, then $xin - yin - 2^{-6} = truncx - trunxy - 2^{-6}$. The theorems are shown in figure 2.

## 4.4. *Generality of the proof procedure*

We have shown that an SRT division circuit can be verified automatically by the theorem prover Analytica. In this section, we discuss the generality of our proof technique.

There are two main approaches to achieving generality with theorem proving techniques: *fully automated methods* and *interactive parameterized proofs*. Our work is in the first class. Only about two pages of input to Analytica are needed to define the circuit and the theorems to be proven. Analytica relies on automatic symbolic algebra algorithms and the Sup-Inf decision procedure to prove the theorems automatically.

Consider the problem of applying our method to another SRT division circuit. If the new circuit differs from Taylor's circuit [22] only in having a different quotient lookup table, it would be sufficient to modify the definition of the the next quotient digit function. Then the proof could be rerun automatically. Similarly, changing the width of the data path or the radix of the quotient digits could be accomplished by simply changing some of the constants in the Analytica input, and rerunning the proof automatically.

In essence, the automatic nature of Analytica implies that if we can obtain a formalization and proof of a circuit, we also obtain an automatic proof procedure that can be applied to related circuits.

*Interactive parameterized proofs* [13] are another approach to achieving generality. The basic idea is to introduce new variables to stand for parameters of the circuit, such as the word size or radix. A combinational function such as the quotient lookup function can also be treated as a circuit parameter. In this approach, the user formulates and proves a general theorem involving the parameterized design. Once the general theorem has been proven, it is ususally not difficult to *instantiate* the parameters with constant values to obtain a result for a particular circuit.

```
(* Main result for the addition case of GALU. *)

Prove[imp[and[xin == -8*x1 + 4*x2 + 2* x3 + lowx,
              yin == -8*y1 + 4*y2 + 2*y3 + lowy,
              or[x1 == 1, x1 == 0],
              or[x2 == 1, x2 == 0],
              or[x3 == 1, x3 == 0],
              or[y1 == 1, y1 == 0],
              or[y2 == 1, y2 == 0],
              or[y3 == 1, y3 == 0],
              0 <= lowx < 2,
              0 <= lowy < 2,
              truncx == 2*x3 + lowx,
              truncy == 2*y3 + lowy,
              truncx + truncy == 4*z1 + 2*z2 + lowz,
              or[z1 == 0, z1 == 1],
              or[z2 == 0, z2 == 1],
              0 <= lowz < 2,
              -2 <= xin + yin < 2],
          xin + yin == -2*z2 + lowz]];

(* Main result for subtraction case of GALU. *)

Prove[imp[and[k == 2^(-6),
              xin == -8*x1 + 4*x2 + 2*x3 + lowx,
              yin == -8*y1 + 4*y2 + 2*y3 + lowy,
              or[x1 == 1, x1 == 0],
              or[x2 == 1, x2 == 0],
              or[x3 == 1, x3 == 0],
              or[y1 == 1, y1 == 0],
              or[y2 == 1, y2 == 0],
              or[y3 == 1, y3 == 0],
              0 <= lowx <= 2-k,
              0 <= lowy <= 2-k,
              truncx == 2*x3 + lowx,
              truncy == 2*y3 + lowy,
              truncx - truncy - k == -4*z1 + 2*z2 + lowz,
              or[z1 == 0, z1 == 1],
              or[z2 == 0, z2 == 1],
              0 <= lowz < 2,
              -2 <= xin - yin - k <= 2],
          xin - yin - k == -2*z2 + lowz]];
```

*Figure 2.* Analytica theorems for the GALU.

One disadvantage of parameterized proofs is that using parameters in the formalization of a circuit can make the resulting theorems more difficult to prove automatically. For example, suppose a linear term of the form $cx$ appears in a formula, where $c$ is a constant related to circuit size and $x$ is a variable. If we replace $c$ by a new variable $y$, then the linear term is replaced with a non linear term, which makes automatic reasoning more difficult. Phenomena such as this make parameterized proofs more difficult to obtain automatically than proofs of the individual instances.

## 5. Conclusion

In this paper, we investigate a radix-4 SRT division algorithm similar to the one used in the Intel Pentium processor. We have built a formal model for the circuit and proven the correctness of the model using our theorem prover Analytica. The main properties that have been proven by Analytica are invariants saying that the value of the partial remainder stays in the correct range, and that the divider only accesses defined entries in the quotient lookup table. These are the most critical properties to verify for an SRT divider, because they imply that the quotient lookup table is correct. As the Pentium error showed, the correctness of a quotient lookup table is difficult to assure without formal verification.

We have also used Analytica to verify a subtle logic optimisation that the circuit designer used in one of the arithmetic units of the divider. Taylor's circuit uses an estimation ALU (GALU) to compute an estimate of the partial remainder on each cycle. The subtlety is that while the signals that must be added or subtracted are 10 bits wide, the arithmetic unit is only an 8-bit ALU. This design can be justified by appealing to invariants on the values of partial remainder and estimated partial remainder and by using some properties of two's complement arithmetic. We have used Analytica to show that the 8-bit ALU produces the correct result.

The main obstacle to wider use of theorem proving techniques for hardware verification is the need for detailed user guidance when using most theorem provers. Therefore, it is significant that Analytica is able to use our formulation of the SRT problem to prove the main correctness invariants of the circuit with only a small amount of guidance. Analytica needs only one user-supplied lemma to prove the main results.

The high degree of automation that we obtained is a consequence of both the theorem prover we used and the way we formulated the problem. In particular, by formulating truncation of signals using linear inequalities, we made it easier to apply automated reasoning using the Sup-Inf algorithm.

One issue in theorem proving is whether to verify parameterized circuit designs or individual instances of circuits. In principle, the advantage of verifying a parameterized design using theorem proving techniques is that the proof effort can be amortized over all the instances of the design. However, proofs of parameterized designs tend to be much harder to automate than proofs of the individual instances. When highly automated proofs, such as the ones in this paper, can be obtained for individual circuits, we feel there is little reason to carry out the more difficult proofs of parameterized families of circuits.

In mechanised deduction, there is a tradeoff between generality and the degree of automation. Interactive theorem provers are usually developed with many possible applications in mind. In contrast, Analytica addresses a specific domain. A problem that must be overcome in any theorem prover is that generality makes it harder to design mechanisms to choose the correct path in a proof. For instance, in a prover with many simplification mechanisms, if the wrong simplification is chosen at a step in a proof, the prover may never find the rest of the proof. The tradeoff between generality and the degree of automation is difficult to avoid.

In other research, we have developed a *word level model checker* [9] that can verify arithmetic circuits. Although word level model checking works extremely well for many circuits, there are still serious restrictions on the application of this technique. For example, it can only handle circuits that maintain the exact value of the data and would not be applicable for a circuit that involves rounding.

Theorem provers, on the other hand, can be applied to a wider range of problems and are particularly useful for reasoning at a high level of abstraction (architectural level verification). For instance, in this paper, we used a theorem prover to show that the division circuit is correct for all word lengths greater than 10 bits. Finite-state methods such as model checking usually verify a circuit only for a single word length. However, circuit verification by theorem proving techniques usually requires some user interaction, while model checking is largely automatic.

In the future, we intend to combine automatic theorem proving and model checking. There has already been some work in this direction [14, 17]. This combination of approaches should make it possible to handle much larger circuits than is currently the case. In proving some property of a circuit, the specification will be decomposed into sub-goals. Each sub-goal is verified using a decision procedure or the model checker. Then the theorem prover is used to combine the proofs of the sub-goals.

## Appendix

### *A.1. Mathematica code for the problem*

```
<< index.all
$RecursionLimit = 2000;

(* the main ALU *)
next[rout] := next[ite[qsign, rin - md, rin + md]];

(* md is a multiple of the divisor *)
md := d qdigit;

(* left shift by 2 bits *)
next[rin] := 4 rout;

(* values from quotient table. *)
next[qdigit] := nextqdigit[4 trout1, qtable[d1]];
next[qsign] := nextqsign[4 trout1, qtable[d1]];
```

```
(* divisor never changes *)
next[d] := d;

next[n_?NumberQ] := n;

(* compute the quotient word *)
next[qpos] := ite[qsign, 4 qpos + qdigit, 4 qpos];
next[qneg] := ite[qsign, 4 qneg, 4 qneg + qdigit];

(* quotient := qpos - qneg; *)
quotient := qpos - qneg;

next[f_[a___]] := Map[next, f[a]];

(* Each row of the table is a list {b1, b2, b3, b4, b5, b6}.
When b1 <= r < b2, q = -2;
b2 <= r < b3, q = -1;
b3 <= r < b4, q = 0;
b4 <= r < b5, q = 1;
b5 <= r < b6, q = 2;
r < b1, out of table and we set q = -3;
r >= b6, out of table and we set q = 3;

out of table -2     -1     0     1     2    out of table     q
-----------+-----+-----+-----+-----+-----+--------------> rout1
           b1    b2    b3    b4    b5    b6
*)

qtable[1]    := {  -7/2, -13/8, -1/2, 1/2, 3/2,    3};
qtable[9/8]  := {  -7/2,  -7/4, -1/2, 1/2, 7/4,  7/2};
qtable[5/4]  := {    -4,    -2, -3/4, 1/2,   2,    4};
qtable[11/8] := {  -9/2,  -9/4, -3/4, 1/2,   2,    4};
qtable[3/2]  := {  -9/2,  -5/2,   -1, 3/4, 9/4,  9/2};
qtable[13/8] := {    -5,  -5/2,   -1,   1, 5/2,    5};
qtable[7/4]  := { -11/2, -11/4,   -1,   1, 5/2,    5};
qtable[15/8] := { -11/2,    -3,   -1,   1,   3, 11/2};

(* CaseWithDefault is a conditional expression. Each case
is a pair of a boolean expression and a value. The final
element is the default value, which is 3 in this example.
If none of the boolean cases applies, the default value
is returned. *)
```

```
nextqdigit[r_, {b1_, b2_, b3_, b4_, b5_, b6_}] :=
  CaseWithDefault[{(b1 <= r < b2) || (b5 <= r < b6), 2},
                  {(b2 <= r < b3) || (b4 <= r < b5), 1},
                  {(b3 <= r < b4), 0},
                  3];

(* sign of q *)
nextqdigit[r_, {b1_, b2_, b3_, b4_, b5_, b6_}] := (r >= b3);

(* The quantity (rin + 4 quotient d) is multiplied by 4 on each
iteration by left shifting. The hypothesis about rout is needed
because we only defined the value of next[rout]. *)

Prove[imp[rout == ite[qsign, rin - md, rin + md],
          next[rin + 4 quotient d] == 4(rin + 4 quotient d)]];

(* introduce an abbreviation *)

approx[a_, b_, n_] := (a <= b < a + 2^(-n));

(* Since rout1 has 6 bits to the right of the binary point, its
value is a multiple of 2^-6. *)

DeclarePrecision[rout1] := 2^-6;

(* The Hypothesis says that rin1 and md1 are approximations of
rin and md to 6 bits after decimal points. rout1 is the out put
of the small ALU. The conclusion of the theorem is that
rout1 <= rout < rout1 + 2^(-5). This theorem is used as a lemma
in proving the two following theorems. *)

Prove[imp[and[approx[rin1, rin, 6], approx[md1, md, 6],
              rout1 == ite[qsign, rin1 - md1 - 2^(-6), rin1 + md1],
              rout == ite[qsign, rin - md, rin + md]],
          approx[rout1, rout, 5]]];

(* The hypothesis says that rout1 <= rout < rout1 + 2^(-5); d1
is approximation of divisor to 3 bits after the decimal point;
d1 can only have 8 possible values; rout is between -2/3 d and
2/3 d. The conclusion is that next[rout] is between -2/3 d and
2/3 d. *)

DeclarePrecision[trout1] := 2^(-5);
```

```
Prove[imp[and[d1 <= d < d1 + 2^(-3),
              or[d1 == 8/8, d1 == 9/8, d1 == 10/8, d1 == 11/8,
                 d1 == 12/8, d1 == 13/8, d1 == 14/8, d1 == 15/8],
              rout1 <= rout < rout1 + 2^(-5),
              trout1 <= rout1 <= trout1 + 2^(-6),
              -2/3 d <= rout < 2/3 d],
          -2/3 d <= next[rout] < 2/3 d]];

(* The hypothesis says that rout1 <= rout < rout1 + 2^(-5); d1
is approximation of the divisor to 3 bits after the decimal
point; d1 can only have 8 possible values; rout is between
-2/3d and 2/3d. The conclusion is that the quotient can only be
-2, -1, 0, 1, 2. This guanrantees that it is impossible to fall
out of table. *)

Prove[imp[and[approx[rout1, rout, 5], approx[d1, d, 3],
              trout1 <= rout1 <= trout1 + 2^(-6),
              or[d1 == 8/8, d1 == 9/8, d1 == 10/8, d1 == 11/8,
                 d1 == 12/8, d1 == 13/8, d1 == 14/8, d1 == 15/8],
              -2/3 d <= rout < 2/3 d],
          or[next[qdigit] == 0,
             next[qdigit] == 1,
             next[qdigit] == 2]]];
```

## *A.2. An example of the proof*

**Theorem.**

$$\begin{aligned}&approx(rin1, rin, 6) \wedge approx(md1, md, 6) \wedge\\ &rout1 = ite(qsign, rin1 - md1 - 2^{-6}, rin1 + md1) \wedge\\ &rout = ite(qsign, rin - md, rin + md) \Longrightarrow\\ &approx(rout1, rout, 5)\end{aligned}$$

**Proof:**

$$\begin{aligned}&rin1 \le rin \wedge rin < \frac{1}{64} + rin1 \wedge md1 \le d\, qdigit \wedge d\, qdigit < \frac{1}{64} + md1 \wedge\\ &ite(qsign, rout1 = -\frac{1}{64} - md1 + rin1, rout1 = md1 + rin1) \wedge\\ &ite(qsign, rout = -\,(d\, qdigit) + rin, rout = d\, qdigit + rin) \Longrightarrow\\ &rout1 \le rout \wedge rout < \frac{1}{32} + rout1\end{aligned}$$

reduces to

$$-rin + rin1 \leq 0 \wedge -\frac{1}{64} + rin - rin1 < 0 \wedge md1 - d\,qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - md1 + d\,qdigit < 0 \wedge$$
$$ite(qsign, rout1 = -\frac{1}{64} - md1 + rin1, rout1 = md1 + rin1) \wedge$$
$$ite(qsign, rout = -(d\,qdigit) + rin, rout = d\,qdigit + rin) \Longrightarrow$$
$$-rout + rout1 \leq 0 \wedge -\frac{1}{32} + rout - rout1 < 0$$

and split
case 1.1:

$$-rin + rin1 \leq 0 \wedge -\frac{1}{64} + rin - rin1 < 0 \wedge md1 - d\,qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - md1 + d\,qdigit < 0 \wedge$$
$$ite(qsign, rout1 = -\frac{1}{64} - md1 + rin1, rout1 = md1 + rin1) \wedge$$
$$ite(qsign, rout = -(d\,qdigit) + rin, rout = d\,qdigit + rin) \Longrightarrow$$
$$-rout + rout1 \leq 0$$

ite cases
case 1.1.1:

$$-rin + rin1 \leq 0 \wedge -\frac{1}{64} + rin - rin1 < 0 \wedge md1 - d\,qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - md1 + d\,qdigit < 0 \wedge qsign \wedge$$
$$rout1 = -\frac{1}{64} - md1 + rin1 \wedge ite(qsign, rout =$$
$$-(d\,qdigit) + rin, rout = d\,qdigit + rin) \Longrightarrow$$
$$-rout + rout1 \leq 0$$

simplify formula using local context

$$-rin + rin1 \leq 0 \wedge -\frac{1}{64} + rin - rin1 < 0 \wedge md1 - d\,qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - md1 + d\,qdigit < 0 \wedge qsign \wedge$$
$$rout1 = -\frac{1}{64} - md1 + rin1 \wedge rout = -(d\,qdigit) + rin \Longrightarrow$$
$$-rout + rout1 \leq 0$$

$$and(\infty \leq rin1, rin1 \leq -\infty)$$

Proved by SupInf Method

case 1.1.2:

$$-rin + rin1 \leq 0 \wedge -\frac{1}{64} + rin - rin1 < 0 \wedge md1 - d\, qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - md1 + d\, qdigit < 0 \wedge$$
$$rout1 = md1 + rin1 \wedge ite(qsign, rout =$$
$$-(d\, qdigit) + rin, rout = d\, qdigit + rin) \Longrightarrow$$
$$qsign \vee -rout + rout1 \leq 0$$

simplify formula using local context

$$-rin + rin1 \leq 0 \wedge -\frac{1}{64} + rin - rin1 < 0 \wedge md1 - d\, qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - md1 + d\, qdigit < 0 \wedge$$
$$rout1 = md1 + rin1 \wedge rout = d\, qdigit + rin \Longrightarrow$$
$$qsign \vee -rout + rout1 \leq 0$$

$$and(\infty \leq rin1, rin1 \leq -\infty)$$

Proved by SupInf Method

case 1.2:

$$-rin + rin1 \leq 0 \wedge -\frac{1}{64} + rin - rin1 < 0 \wedge md1 - d\, qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - md1 + d\, qdigit < 0 \wedge$$
$$ite(qsign, rout1 = -\frac{1}{64} - md1 + rin1, rout1 = md1 + rin1) \wedge$$
$$ite(qsign, rout = -(d\, qdigit) + rin, rout = d\, qdigit + rin) \wedge$$
$$-rout + rout1 \leq 0 \Longrightarrow$$
$$-\frac{1}{32} + rout - rout1 < 0$$

simplify formula using local context

$$-rin + rin1 \leq 0 \wedge -\frac{1}{64} + rin - rin1 < 0 \wedge md1 - d\, qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - md1 + d\, qdigit < 0 \wedge$$
$$ite(qsign, rout1 = -\frac{1}{64} - md1 + rin1, rout1 = md1 + rin1) \wedge$$
$$ite(qsign, rout = -(d\, qdigit) + rin, rout = d\, qdigit + rin) \Longrightarrow$$
$$-\frac{1}{32} + rout - rout1 < 0$$

ite cases
case 1.2.1:

$$-rin + rinl \leq 0 \wedge -\frac{1}{64} + rin - rinl < 0 \wedge mdl - d\,qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - mdl + d\,qdigit < 0 \wedge qsign \wedge$$
$$routl = -\frac{1}{64} - mdl + rinl \wedge ite(qsign, rout =$$
$$-(d\,qdigit) + rin, rout = d\,qdigit + rin) \Longrightarrow$$
$$-\frac{1}{32} + rout - routl < 0$$

simplify formula using local context

$$-rin + rinl \leq 0 \wedge -\frac{1}{64} + rin - rinl < 0 \wedge mdl - d\,qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - mdl + d\,qdigit < 0 \wedge qsign \wedge$$
$$routl = -\frac{1}{64} - mdl + rinl \wedge rout = -(d\,qdigit) + rin \Longrightarrow$$
$$-\frac{1}{32} + rout - routl < 0$$

$$and(\infty \leq rinl, rinl \leq -\infty)$$

Proved by SupInf Method
case 1.2.2:

$$-rin + rinl \leq 0 \wedge -\frac{1}{64} + rin - rinl < 0 \wedge mdl - d\,qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - mdl + d\,qdigit < 0 \wedge$$
$$routl = mdl + rinl \wedge ite(qsign, rout =$$
$$-(d\,qdigit) + rin, rout = d\,qdigit + rin) \Longrightarrow$$
$$qsign \vee -\frac{1}{32} + rout - routl < 0$$

simplify formula using local context

$$-rin + rinl \leq 0 \wedge -\frac{1}{64} + rin - rinl < 0 \wedge mdl - d\,qdigit \leq 0 \wedge$$
$$-\frac{1}{64} - mdl + d\,qdigit < 0 \wedge$$
$$routl = mdl + rinl \wedge rout = d\,qdigit + rin \Longrightarrow$$

$$qsign \vee -\frac{1}{32} + rout - rout1 < 0$$

$$and(\infty \leq rin1, rin1 \leq -\infty)$$

Proved by SupInf Method □

## Acknowledgment

We thank Nikolaj Bjorner for calling our attention to a technical problem.

This research was sponsored in part by the National Science Foundation under Grant No. CCR-9217549, by the Semiconductor Research Corporation under Contract No. 94-DJ-294, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under Grant No. F33615-93-1-1330.

## Notes

1. Here we consider the case of two's complement notation, but it is straightforward to give an inequality for truncation of numbers in sign-magnitude notation. Truncation of sign-magnitude numbers always gives a result with the same sign but smaller absolute value.

## References

1. R. Alur and T. Henzinger (Eds.), *Computer-Aided Verification (CAV '96)*, volume 1102 of Lecture Notes in Computer Science, Springer-Verlag, 1996.
2. APT Data Services, "Pentium bug fiasco costs Intel dear," *Computer Business Review*, Jan. 1995.
3. D.E. Atkins, "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Transactions on Computers*, Vol. C-17, No. 10, pp. 925–934, Oct. 1968.
4. W.W. Bledsoe, "The UT natural deduction prover," Technical Report ATP-17B, Mathematical Department, University of Texas at Austin, TX, 1983.
5. W.W. Bledsoe, P. Bruell, and R. Shostak, "A prover for general inequalities," Technical Report ATP-40A, Mathematical Department, University of Texas at Austin, TX, 1979.
6. R.S. Boyer and J.S. Moore, *A Computational Logic Handbook*, Academic Press, 1988.
7. E.M. Clarke and S.M. German, Personal communication to H. Ruess, N. Shankar, and M.K. Srivas, 1995.
8. E.M. Clarke, S.M. German, and X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques," in R. Alur and T. Henzinger (Eds.), *Computer-Aided Verification (CAV '96)*, volume 1102 of Lecture Notes in Computer Science, Springer-Verlag, 1996.
9. E.M. Clarke, M. Khaira, and X. Zhao, "Word level symbolic model checking—Avoiding the pentium FDIV error," *Design Automation Conference*, June 1996.
10. E.M. Clarke and X. Zhao, "Analytica: A theorem prover for mathematica," *The Journal of Mathematica*, Vol. 3, No. 1, 1993.
11. J.H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper & Row, 1986.
12. S.M. German, "Verification of arithmetic hardware using a symbolic algebra system," *Lecture Notes*, March 1995.
13. S.M. German and Y. Wang, "Verification of parameterized hardware designs," in *Proceedings of International Conference on Computer Design*, 1985.
14. J. Joyce and C. Seger, "The HOL-Voss system: Model-checking inside a general-purpose theorem prover," in *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications, HUG '93, LNCS 780*, Springer-Verlag, 1993.

15. J. O'Leary, M. Leeser, J. Hickey, and M. Aagaard, "Non-restoring integer square root: A case study in design by principled optimization," in *Proceedings of the Theorem Provers in Circuit Design '94, LNCS 901*, Springer-Verlag, 1995.
16. V. Pratt, "Anatomy of the Pentium bug," in *Proceedings of TAPSOFT '95: Theory and Practice of Software Development, LNCS 915*, Springer-Verlag, 1995.
17. S. Rajan, N. Shankar, and M.K. Srivas, "An integration of model checking with automated proof checking," in *Proceedings of the Seventh Workshop on Computer-Aided Verification*, 1995.
18. H. Ruess, N. Shankar, and M.K. Srivas, "Modular verification of SRT division," preliminary version in [1], final version in this journal.
19. E. Sacks, "Hierarchical inequality reasoning," Technical Report, MIT Laboratory for Computer Science, 1987.
20. H.P. Sharangpani and M.L. Barton, "Statistical analysis of floating point flaw in the Pentium processor (1994)," Technical Report, Intel Corporation, Nov. 1994.
21. R. Shostak, "On the sup-inf method for proving Presburger formulas," *Journal of the Association for Computing Machinery*, Vol. 24, pp. 529–543, 1977.
22. G.S. Taylor, "Compatible hardware for division and square root," in *Proceedings of the 5th IEEE Symposium on Computer Arithmetic*, May 1981.
23. 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*, Vol. 4, pp. 5–31, Jan. 1994.
24. S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Wolfram Research Inc., 1988.