A Mechanically Verified Commercial SRT Divider

David M. Russinoff

April 9, 2009

Abstract

We present a proof of correctness of a commercial implementation of the Sweeney-Robertson-Tocher (SRT) division algorithm, namely the integer divider of the AMD processor code-named "Llano". The register-transfer logic (RTL) design of the divider and its behavioral specification are both formalized in the ACL2 logic; the proof has been formally checked by the ACL2 prover. The complexity of the problem is managed by modeling the design at successively lower levels of abstraction, beginning with the SRT algorithm and ending with the RTL module. This approach is contrasted with earlier published work on this problem, which addresses only the high-level algorithm.

1 Introduction

The Sweeney-Robertson-Tocher (SRT) division algorithm [9, 13], owing to its susceptibility to efficient hardware implementation, is ubiquitous in contemporary microprocessor designs. It is also notoriously prone to implementation error. Analysis of the algorithm's most celebrated incarnation, the defective FDIV circuitry of the original Intel Pentium floating-point unit [7], suggests that thorough verification of an SRT divider by testing alone is a practical impossibility.

This development has been a boon to the enterprise of formal hardware verification. One early response to the 1994 revelation of the Pentium bug was Bryant's BDD-based analysis [2], which established a critical invariant of an SRT circuit but was limited by the practical constraints of the model-theoretic approach. More complete results were subsequently achieved with the use of mechanical theorem provers by Kapur and Subramanian [5]; Ruess, Shankar, and Srivas [8]; and Clarke, German, and Zhou [3]. All of these efforts shared the goal of demonstrating the effectiveness of a particular prover in exposing a specific bug, and consequently focused on the relevant aspects of the underlying algorithm. Moreover, although developed independently, all three were coincidentally based on the same execution model, a high-level circuit design proposed in 1981 by Taylor [12].

A different objective is pursued in the work reported here: a comprehensive machine-checked proof of correctness of a commercial hardware design. The object of investigation is a register-transfer logic (RTL) model of a radix-4 SRT integer divider, to be implemented as a component of the AMD processor code-named "Llano". The theorem prover used in this project is ACL2 [1].

The required behavior of the module is concisely specified in terms of the integer values X and Y (divisor and dividend) represented by the primary data inputs and the corresponding values Q and R (quotient and remainder) of the data outputs. Under suitable input constraints, the following relations must be satisfied:

(1) $Y = QX + R$;

(2) $|R| < |X|$;

(3) Either $R = 0$, or R and X have the same sign.

Regrettably (from a verification perspective), the simplicity of this behavioral specification is not reflected in the design. In contrast to Taylor's circuit, which uses only five state-holding registers, the divider of the Llano processor uses fifty-six. In order to address this complexity, the proof is divided into four parts, which model the design at successively lower levels of abstraction.

At the highest level, as discussed in Section 2, we establish the essential properties of the underlying SRT algorithm. Our description of the algorithm is based on an unspecified radix, 2^r. In the case of interest, we have $r = 2$, which means that two quotient bits are generated per cycle. The main result of this section pertains to the iterative phase of the computation, which generates the sequences of partial remainders $p_0, \ldots, p_n$, quotient digits $m_1, \ldots, m_n$, and resulting partial quotients $Q_0, \ldots, Q_n$. We also address several relevant issues that are ignored in the proofs cited above: (1) prescaling of the divisor and dividend and postscaling of the remainder; (2) determination of the required number n of iterations, which depends on the relative magnitudes of the operands; (3) incremental ("on-the-fly") computation of the quotient, which involves the integration of positive and negative quotient digits; and (4) derivation of the final remainder and quotient R and Q, as specified above, from the results R' and Q' of the iteration, which are characterized by $Y = Q'X + R'$ and $|R'| \leq |X|$.

In the radix-4 case, the quotient digits are confined to the range $-3 \leq m_k \leq 3$. Each m_k is read from a table of $4 \times 32 = 128$ entries according to indices derived from the normalized divisor d and the previous partial remainder p_{k-1}, and is used to compute the next partial remainder by the recurrence formula $p_k = 4p_{k-1} - m_k d$. At the second level of abstraction, in Section 3, we present the actual table used in our implementation, which was adapted from the IBM z990 [4], and prove that it preserves the invariant $|p_k| \leq |d|$.

At the third level, the algorithm is implemented in XFL, a simple formal language developed at AMD for the specification of the AMD64 instruction set architecture. XFL is based on unbounded integer and arbitrary precision rational data types and combines the basic constructs of C with the logical bit vector operations of Verilog, in which AMD RTL designs are coded. The XFL encodings of the lookup table and the divider are displayed in Appendices A and B. Like most XFL programs, this code was automatically generated from a hand-coded C++ program, which has been subjected to testing for the purpose of validating the model.

The XFL model is significantly smaller than the RTL, which consists of some 150 kilobytes of Verilog code, but it is designed to perform the same sequence of register-transfer-level operations while avoiding low-level implementation concerns. Thus, much of the complexity of the design is captured at this third level, including several essential features that are absent from higher-level models such as Taylor's circuit specification:

(1) A hardware implementation of Taylor's model, which computes an explicit representation of the partial remainder on each iteration, would require a time-consuming full-width carry-propagate adder, resulting in a prohibitively long cycle

time. In contrast, a typical contemporary commercial implementation such as the Llano divider stores the remainder in a redundant form, which may be computed by a much faster carry-save adder. A single full-width addition is then performed at the end of the iterative phase.

(2) The derivation of the final results R and Q from the intermediate values R' and Q' involves consideration of the special cases $R' = 0$ and $R' = \pm X$. Timing considerations dictate that these conditions be detected in advance of the full-width addition that produces R'. This requires special logic for predicting cancellation.

(3) The module is also responsible for detecting overflow, i.e., a quotient that is too large to be represented in the target format. This involves an analysis that is performed concurrently with the final computation of the quotient.

Each of these complications introduces a possible source of design error that cannot be ignored. In Section 4, we present a complete proof of the claim that the algorithm is correctly implemented by the XFL model.

The lowest level of abstraction to be considered is that of the RTL itself. The proof of equivalence between the RTL and XFL models represents a significant portion of the overall effort, involving the analysis of a complex state machine, innumerable timing and scheduling issues, and various other implementation concerns. However, this part of the proof would be of relatively little interest to a general readership; moreover, neither space nor proprietary confidentiality allows its inclusion here.

Thus, the purpose of this paper is an exposition of the proof of correctness of the Llano divider as represented by the XFL model. The presentation is confined to standard mathematical notation, avoiding any obscure special-purpose formalism, but assumes familiarity with the general theory of bit vectors and logical operations, making implicit use of the results found in [10]. Otherwise, the proof is self-contained and surveyable, with one exception: Lemma 3.1, which provides a set of inequalities that are satisfied by the entries of the lookup table, involves machine-checked computation that is too extensive to be carried out by hand.

We emphasize, however, that a comprehensive statement of correctness of the RTL module itself has been formalized in the logic of ACL2 and its proof has been thoroughly checked with the ACL2 prover. This includes a formalization of the proof presented here, along with a detailed proof of equivalence between the XFL and RTL models. For this purpose, the XFL model was re-coded directly in ACL2 and the RTL module was translated to ACL2 by a tool that was developed for this purpose [11]. Thus, the validity of the proof depends only on the semantic correctness of the Verilog-ACL2 translator and the soundness of the ACL2 prover, both of which have been widely tested.

While the ultimate objective of formal verification is a proof of correctness, its utility may best be demonstrated through the exposure of bugs that might otherwise have gone undetected. In the present case, three bugs that had already survived extensive directed testing were found through our analysis, at three distinct levels of abstraction. Once identified, all three were readily corrected.

First, an entry of the lookup table (the one in the upper right corner of Figure 1) was missing from the original design. This was strikingly similar to the original Pentium bug insofar as the entry was thought to be inaccessible. The error was not exposed by initial directed testing, which was designed to hit all accessible table entries on the first iteration. Subsequent analysis and testing revealed that the entry can be reached, but only after nine iterations.

A second problem was detected in connection with the same table entry. If the partial remainder is close to -2, it appears to be possible (although no test case has yet been constructed to confirm this possibility), as a result of the two's complement encoding scheme, for the approximation derived from the redundant representation to be close to $+2$ instead. The original design did not account for this occurrence, which would have resulted in a quotient digit with the wrong sign.

Finally, a timing problem was detected in the RTL implementation, related to the possible cancellation and re-transmission of the divisor input. This issue, which of course is not reflected in the algorithm or the XFL encoding, illustrates the inadequacy of a correctness proof based solely on a high-level design model.

2 SRT Division

The description of a division algorithm is typically simplified by interpreting its parameters as fractions. Thus, our presentation begins with a normalized representation of the divisor X,

$$d = 2^{-expo(X)}X,$$

where $expo(X)$ is the integer defined by

$$2^{expo(X)} \leq |X| < 2^{expo(X)+1},$$

and consequently, $1 \leq |d| < 2$. The dividend Y is similarly shifted to produce the initial partial remainder,

$$p_0 = 2^{-expo(X)-rn}Y,$$

where 2^r is the underlying radix of the computation and n, the number of iterations to be performed, is chosen to ensure that $|p_0| \leq |d|$.

On each iteration, the current partial remainder p_{k-1} is shifted by r bits and an appropriate multiple of d is subtracted to form the next partial remainder,

$$p_k = 2^r p_{k-1} - m_k d,$$

where the multiplier m_k contributes the accumulated quotient. The invariant $|p_k| \leq |d|$ is guaranteed by selecting m_k from the interval

$$\frac{2^r p_{k-1}}{d} - 1 \leq m_k \leq \frac{2^r p_{k-1}}{d} + 1.$$

For further motivation, refer to [6].

Lemma 2.1 *Let X, Y, r, and n be integers such that $X \neq 0$, $r > 0$, $n \geq 0$, and $|Y| \leq 2^{rn}|X|$. Let $d = 2^{-expo(X)}X$, $p_0 = 2^{-expo(X)-rn}Y$, and $Q_0 = 0$. For $k = 1, \ldots, n$, let*

$$p_k = 2^r p_{k-1} - m_k d$$

and

$$Q_k = 2^r Q_{k-1} + m_k,$$

where m_k is an integer such that if $|p_{k-1}| \leq |d|$, then $|p_k| \leq |d|$. Let $R = 2^{expo(X)}p_n$ and $Q = Q_n$. Then $Y = QX + R$ and $|R| \leq |X|$.

PROOF: Clearly, $1 \leq |d| < 2$, and

$$|p_0| = 2^{-expo(X)-rn}|Y| \leq 2^{-expo(X)}|X| = |d|.$$

It follows by induction that $|p_k| \leq |d|$ for all $k \leq n$. We shall also show by induction that

$$p_k = 2^{rk}p_0 - Q_k d.$$

The claim clearly holds for $k = 0$, and for $0 < k \leq n$,

$$\begin{aligned} p_k &= 2^r p_{k-1} - m_k d \\ &= 2^r \left(2^{r(k-1)}p_0 - Q_{k-1}d\right) - m_k d \\ &= 2^{rk}p_0 - (2^r Q_{k-1} + m_k)d \\ &= 2^{rk}p_0 - Q_k d. \end{aligned}$$

In particular,

$$p_n = 2^{rn}p_0 - Q_n d,$$

and

$$Y = 2^{expo(X)}2^{rn}p_0 = 2^{expo(X)}(Q_n d + p_n) = QX + R,$$

where $|R| = |2^{expo(X)}p_n| \leq |2^{expo(X)}d| = |X|$. □

A quotient and remainder that satisfy the conclusion of Lemma 2.1 may be easily adjusted to satisfy the specification stated in Section 1.

Lemma 2.2 *Let X, Y, Q', and R' be integers such that $X \neq 0$, $|R'| \leq |X|$, and $Y = Q'X + R'$. Let R and Q be defined as follows:*

(a) If $|R'| < |X|$ and either $R' = 0$ or $sgn(R') = sgn(Y)$, then $R = R'$ and $Q = Q'$;

(b) If (a) does not apply and $sgn(R') = sgn(X)$, then $R = R' - X$ and $Q = Q' + 1$;

(c) If (a) does not apply and $sgn(R') \neq sgn(X)$, then $R = R' + X$ and $Q = Q' - 1$.

Then $Y = QX + R$, $|R| < |X|$, and either $R = 0$ or $sgn(R) = sgn(Y)$.

PROOF: We consider the three cases separately:

(a) In this case, the conclusion holds trivially.

(b) In this case,

$$Y = Q'X + R' = (Q' + 1)X + (R' - X) = QX + R$$

and $sgn(R') = sgn(X)$. If $|R'| = |X|$, then $R' = X$ and $R = 0$. Otherwise, we must have $Y \neq 0$ and $sgn(R') \neq sgn(Y)$. Since $|R'| < |X|$, $|R| = |R' - X| = |X| - |R'| < |X|$. Moreover, $sgn(R) \neq sgn(R')$, which implies $sgn(R) = sgn(Y)$.

(c) Here we have

$$Y = Q'X + R' = (Q' - 1)X + (R' + X) = QX + R$$

and $sgn(R') \neq sgn(X)$. If $|R'| = |X|$, then $R' = -X$ and $R = 0$. Otherwise, $Y \neq 0$ and $sgn(R') \neq sgn(Y)$. Since $|R'| < |X|$, $|R| = |R' + X| = |X| - |R'| < |X|$. Moreover, $sgn(R) \neq sgn(R')$, which implies $sgn(R) = sgn(Y)$. □

In an SRT implementation, the multiplier m_k of Lemma 2.1 represents a sequence of r bits that are appended to the quotient during the k^{th} iteration. Although not required for the proof of the lemma, it may be assumed that in practice, $|m_k| < 2^r$. In particular, in our radix-4 implementation, we have $-3 \leq m_k \leq 3$. This provides a bound on the partial quotients.

Lemma 2.3 *Let $Q_0 = 0$ and for $k = 1, \ldots, n$, let $Q_k = 4Q_{k-1} + m_k$, where $-3 \leq m_k \leq 3$. Then $|Q_k| < 4^k$.*

PROOF: By induction,

$$|Q_k| = |4Q_{k-1} + m_k| \leq |4Q_{k-1}| + |m_k| < 4\left(4^{k-1} - 1\right) + 4 = 4^k. \ \Box$$

If we could guarantee that $m_k \geq 0$, then we could maintain a bit vector encoding of the quotient simply by shifting in two bits at each step. In order to accommodate $m_k < 0$ without resorting to a full subtraction, and simultaneously to provide an efficient implementation of Lemma 2.2, we adopt a scheme that involves three separate bit vectors representing the values Q_k, $Q_k - 1$, and $Q_k + 1$. The following lemma will be used in Section 4 to compute the final quotient. Note that each step in the computation may be implemented as a simple two-bit shift.

Lemma 2.4 *Let $Q_0 = 0$ and for $k = 1, \ldots, n$, let $Q_k = 4Q_{k-1} + m_k$, where $-3 \leq m_k \leq 3$. Let $N > 0$. We define three sequences of bit vectors, E_k, E_k^-, and E_k^+, all of width N, as follows: $E_0 = 0$, $E_0^- = 2^N - 1$, $E_0^+ = 1$, and for $k = 1, \ldots, n$,*

$$E_k = \begin{cases} (4E_{k-1} + m_k)[N-1:0] & \text{if } m_k \geq 0 \\ (4E_{k-1}^- + m_k + 4)[N-1:0] & \text{if } m_k < 0, \end{cases}$$

$$E_k^- = \begin{cases} (4E_{k-1} + m_k - 1)[N-1:0] & \text{if } m_k > 0 \\ (4E_{k-1}^- + m_k + 3)[N-1:0] & \text{if } m_k \leq 0, \end{cases}$$

and

$$E_k^+ = \begin{cases} (4E_{k-1} + m_k + 1)[N-1:0] & \text{if } -1 \leq m_k \leq 2 \\ (4E_{k-1}^- + m_k + 5)[N-1:0] & \text{if } m_k < -1 \\ (4E_{k-1}^+)[N-1:0] & \text{if } m_k = 3. \end{cases}$$

Then for $k = 0, \ldots, n$,

$$E_k = Q_k[N-1:0],$$

$$E_k^- = (Q_k - 1)[N-1:0],$$

and

$$E_k^+ = (Q_k + 1)[N-1:0].$$

PROOF: The claim holds trivially for $k = 0$. In the inductive step, there are seven equations to consider. For example, if $m_k < -1$, then

$$\begin{aligned} E_k^+ &= (4E_{k-1}^- + m_k + 5)[N-1:0] \\ &= (4(Q_{k-1} - 1)[N-1:0] + m_k + 5)[N-1:0] \\ &= (4(Q_{k-1} - 1) + m_k + 5)[N-1:0] \\ &= (4Q_{k-1} + m_k + 1)[N-1:0] \\ &= (Q_k + 1)[N-1:0]. \end{aligned}$$

The other six cases are handled similarly. □

The implementation is also responsible for supplying the integer n of Lemma 2.1, which is required to satisfy $|Y| \leq 2^{2n}|X|$ and represents the number of iterations to be performed. This may be accomplished by establishing an upper bound on the difference $expo(Y) - expo(X)$:

Lemma 2.5 *Let X, Y, and B be integers such that $X \neq 0$ and $expo(Y) - expo(X) \leq B$. Let*

$$n = \begin{cases} \lfloor \frac{B}{2} \rfloor + 1 & \text{if } B \geq 0 \\ 0 & \text{if } B < 0. \end{cases}$$

Then $|Y| < 2^{2n}|X|$.

PROOF: If $B \geq 0$, then

$$2n = 2\left(\left\lfloor \frac{B}{2} \right\rfloor + 1\right) \geq 2\left(\frac{B-1}{2} + 1\right) = B + 1,$$

so that $2n + expo(X) \geq B + 1 + expo(X) \geq expo(Y) + 1$ and

$$2^{2n}|X| \geq 2^{2n+expo(X)} \geq 2^{expo(Y)+1} > |Y|.$$

But if $B < 0$, then $expo(Y) < expo(X)$ and

$$|Y| < 2^{expo(Y)+1} \leq 2^{expo(X)} \leq |X| = 2^{2n}|X|. \;\square$$

The most intensive computation performed in the execution of the algorithm is that of the partial remainder, $p_k = 4p_{k-1} + m_k d$. In order for this to be completed in a single cycle, p_k is represented in a redundant form consisting of two bit vectors. Since $|m_k| \leq 3$, the term $m_k d$ is conveniently represented by up to two vectors corresponding to $\pm d$ and $\pm 2d$, depending on m_k. Thus, the computation of p_k is implemented as a two-bit shift (multiplication by 4) of p_{k-1} followed by a 4–2 compression. The details are deferred to Section 4.

The most challenging task is the computation of the quotient digit m_k. This is the subject of the next section.

3 Quotient Digit Selection

In this section, we define a process for computing the quotient bits m_k of Lemma 2.1 and prove that the invariant $|p_k| \leq |d|$ is preserved. The problem may be formulated as follows:

> Given rational numbers d and p such that $1 \leq |d| < 2$ and $|p| \leq |d|$, find an integer m such that $-3 \leq m \leq 3$ and $|4p - dm| \leq |d|$.

We may restrict our attention to the case $d > 0$, since the inequalities in the above objective are unaffected by reversing the signs of both d and m. Thus, we have $1 \leq d < 2$ and $-2 < p < 2$. These constraints determine a rectangle in the dp-plane as displayed in Figure 1, which is adapted from [4]. The rectangle is partitioned into an array of rectangles of width $\frac{1}{4}$ and height $\frac{1}{8}$. The columns and rows of the array are numbered with indices i and j, respectively, where $0 \leq i < 4$ and $0 \leq j < 32$. Let R_{ij} denote the rectangle in column i and row j, and let (δ_i, π_j) be its lower left vertex. Thus,

$$R_{ij} = \left\{ (d,p) \mid \delta_i \leq d < \delta_i + \frac{1}{4} \text{ and } \pi_j \leq p < \pi_j + \frac{1}{8} \right\}.$$

The numbering scheme is designed so that if $(d,p) \in R_{ij}$, then i comprises the leading two bits of the fractional part of d, and j comprises the leading 5 bits of the two's complement representation of p.

The contents of the rectangles of Figure 1 represent a function

$$m = \phi(i,j),$$

which is defined formally in Appendix A and may be implemented as a table of $4 \times 32 = 128$ entries. For a given pair (d,p), we derive an approximation (δ_i, π_j), which determines the arguments of ϕ to be used to compute the corresponding value of m. Ideally, this approximation would be simply determined by the rectangle R_{ij} that contains (d,p), i.e., i and j would be derived by extracting the appropriate bits of d and p. Since our implementation generates the encoding of d explicitly, d may indeed be approximated in this manner. Thus, $i = d[-1:-2] = \lfloor 4(d-1) \rfloor$, which yields

$$\delta_i \leq d < \delta_i + \frac{1}{4}.$$

On the other hand, as noted in Section 2, p is represented redundantly as a sum of two vectors. The index j may be derived by adding the high-order bits of these vectors, but as a consequence of this scheme, as we shall see in Section 4, instead of the optimal range of $\frac{1}{8}$, the accuracy of π_j is given by

$$\pi_j \leq p < \pi_j + \frac{1}{4}.$$

Thus, in geometric terms, we may assume that (d,p) is known to lie within the square S_{ij} formed as the union of the rectangle R_{ij} and the rectangle directly above it:

$$S_{ij} = \left\{ (d,p) \mid \delta_i \leq d < \delta_i + \frac{1}{4} \text{ and } \pi_j \leq p < \pi_j + \frac{1}{4} \right\}.$$

We would like to show that if $(d,p) \in S_{ij}$ and $m = \phi(i,j)$, then $|4p - dm| \leq d$, or equivalently,

$$\frac{m-1}{4} \leq \frac{p}{d} \leq \frac{m+1}{4}.$$

We first present an informal argument, which will then be formalized and proved analytically.

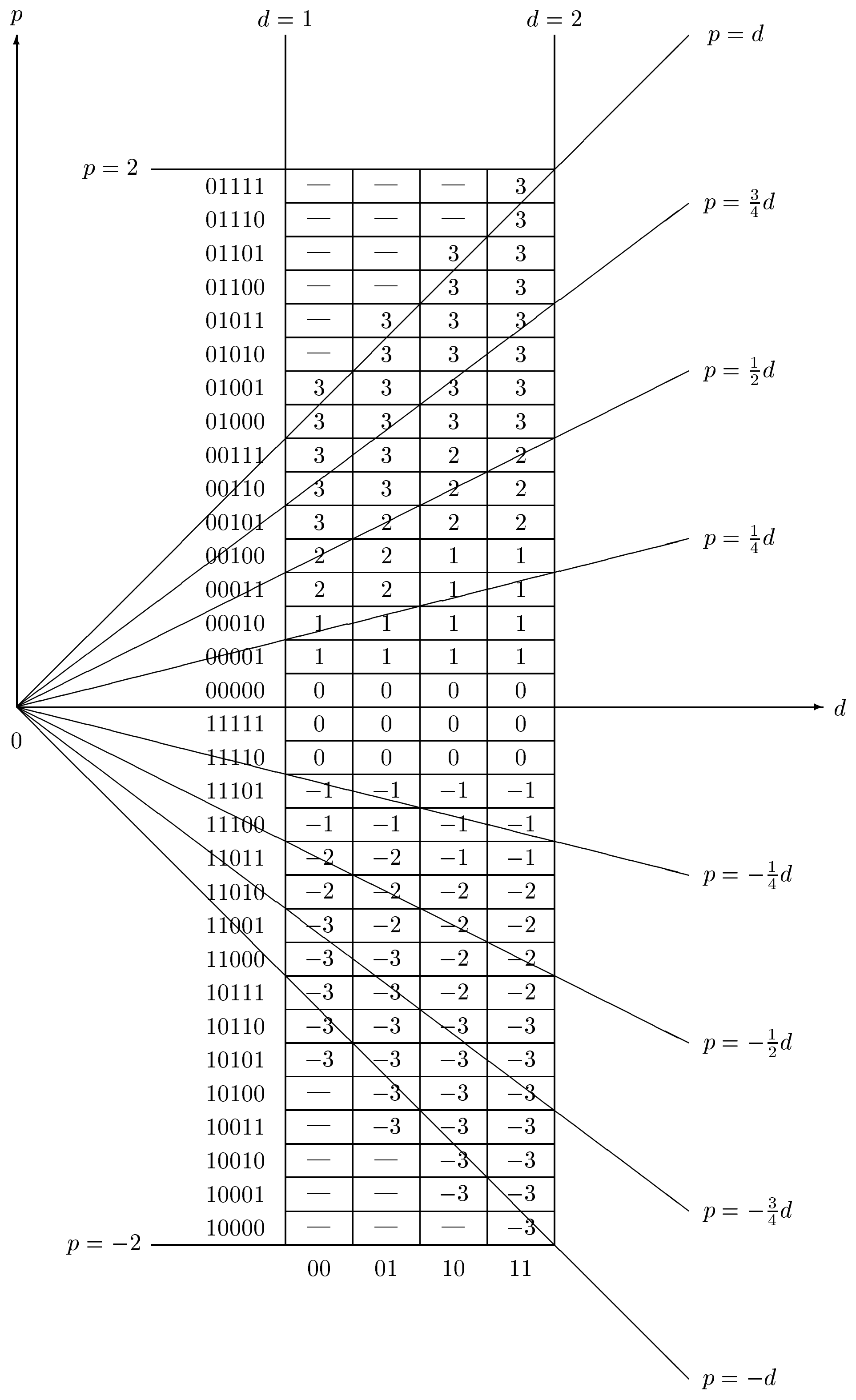
The definition of ϕ is driven by the following observations:

(1) Since $|p| \leq d$, (d,p) lies between the lines $p = d$ and $p = -d$. Therefore, if S_{ij} lies entirely above the line $p = d$, or entirely below the line $p = -d$, then m is inconsequential and left undefined. In all other cases, m is defined.

(2) Since $p \leq d$, the upper bound

$$\frac{p}{d} \leq \frac{m+1}{4}$$

is satisfied trivially if $m = 3$. In order to guarantee that this bound holds generally, it suffices to ensure that if $m \neq 3$, then S_{ij} lies below the line $p = \frac{(m+1)d}{4}$.



	00	01	10	11
01111	—	—	—	3
01110	—	—	—	3
01101	—	—	3	3
01100	—	—	3	3
01011	—	3	3	3
01010	—	3	3	3
01001	3	3	3	3
01000	3	3	3	3
00111	3	3	2	2
00110	3	3	2	2
00101	3	2	2	2
00100	2	2	1	1
00011	2	2	1	1
00010	1	1	1	1
00001	1	1	1	1
00000	0	0	0	0
11111	0	0	0	0
11110	0	0	0	0
11101	-1	-1	-1	-1
11100	-1	-1	-1	-1
11011	-2	-2	-1	-1
11010	-2	-2	-2	-2
11001	-3	-2	-2	-2
11000	-3	-3	-2	-2
10111	-3	-3	-2	-2
10110	-3	-3	-3	-3
10101	-3	-3	-3	-3
10100	—	-3	-3	-3
10011	—	-3	-3	-3
10010	—	—	-3	-3
10001	—	—	-3	-3
10000	—	—	—	-3

Figure 1: SRT Table

(3) Similarly, since $p \geq -d$, the lower bound

$$\frac{p}{d} \geq \frac{m-1}{4}$$

is satisfied trivially if $m = -3$. In order to guarantee that this bound holds generally, it suffices to ensure that if $m \neq -3$, then S_{ij} lies above the line $p = \frac{(m-1)d}{4}$.

It is easily verified by inspection of Figure 1 that in all cases in which m is defined, the conditions specified by (2) and (3) are satisfied and consequently, the desired inequality holds. It should also be noted that in some cases, there is a choice between two acceptable values of m. If S_{ij} lies within the region bounded by $p = \frac{m}{4}d$ and $p = \frac{m+1}{4}d$, where $-3 \leq m \leq 2$, then the inequality is satisfied by both m and $m+1$. For example, although we have assigned 3 as the value of $\phi(11\text{b}, 01000\text{b})$, since $S_{11,01000}$ lies between $p = \frac{1}{2}d$ and $p = \frac{3}{4}d$, we could have chosen 2 instead.

The first step toward formalization is to express the conditions listed above in precise analytical terms:

(1) S_{ij} lies entirely above the line $p = d$ if and only if its lower right vertex, $(\delta_i + \frac{1}{4}, \pi_j)$, lies on or above that line, a condition expressed by the inequality

$$\pi_j \geq \delta_i + \frac{1}{4}.$$

The condition that S_{ij} lies entirely below the line $p = -d$ is similarly determined by the location of its upper right vertex, $(\delta_i + \frac{1}{4}, \pi_j + \frac{1}{4})$, and is expressed by the inequality

$$\pi_j \leq -\left(\delta_i + \frac{1}{4}\right) - \frac{1}{4} = -\delta_i - \frac{1}{2}.$$

Thus, $m = \phi(i,j)$ is defined if and only if neither of these inequalites holds, i.e.,

$$-\delta_i - \frac{1}{2} < \pi_j < \delta_i + \frac{1}{4}.$$

(2) The maximum value of the quotient $\frac{d}{p}$ in S_{ij} occurs at either the upper left or the upper right vertex, depending on the sign of their common p-coordinate, $\pi_j + \frac{1}{4}$. Thus, S_{ij} lies below the line $p = \frac{(m+1)d}{4}$ if and only if both vertices lie on or below the line, i.e,

$$\frac{\pi_j + \frac{1}{4}}{\delta_i} = \frac{4\pi_j + 1}{4\delta_i} \leq \frac{m+1}{4}$$

and

$$\frac{\pi_j + \frac{1}{4}}{\delta_i + \frac{1}{4}} = \frac{4\pi_j + 1}{4\delta_i + 1} \leq \frac{m+1}{4}.$$

(3) The minimum value of $\frac{d}{p}$ in S_{ij} occurs at either the lower left or the lower right vertex, depending on the sign of π_j. Thus, S_{ij} lies above the line $p = \frac{(m-1)d}{4}$ if and only if both vertices lie on or above the line, i.e,

$$\frac{\pi_j}{\delta_i} \geq \frac{m-1}{4}$$

and
$$\frac{\pi_j}{\delta_i + \frac{1}{4}} = \frac{4\pi_j}{4\delta_i + 1} \geq \frac{m-1}{4}.$$

We shall also require analytical expressions for δ_i and π_j as functions of i and j. The definition of δ_i is trivial:

Definition 3.1 *For each integer i such that $0 \leq i < 4$,*
$$\delta_i = 1 + \frac{i}{4}.$$

Since j is the five-bit two's complement representation of the signed integer $8\pi_j$, we have the following definition, in which the function $SgndIntVal(w, x)$ computes the value represented by a bit vector x with respect to a signed integer format of width w:

Definition 3.2 *For each integer j such that $0 \leq j < 32$,*
$$\pi_j = SgndIntVal(5, j) = \begin{cases} \frac{j}{8} & \text{if } j < 16 \\ \frac{j}{8} - 32 & \text{if } j \geq 16. \end{cases}$$

The formal statement of correctness of ϕ appears below as Lemma 3.2. The constraints on ϕ that were derived above are required in the proof. These are summarized in Lemma 3.1, which is proved by straightforward exhaustive computation.

Lemma 3.1 *Let i and j be integers, $0 \leq i < 4$ and $0 \leq j < 32$. Assume that $-\delta_i - \frac{1}{2} < \pi_j < \delta_i + \frac{1}{4}$ and let $m = \phi(i, j)$.*

(a) If $m \neq 3$, then $max\left(\frac{4\pi_j+1}{4\delta_i}, \frac{4\pi_j+1}{4\delta_i+1}\right) \leq \frac{m+1}{4}$;

(b) If $m \neq 3$, then $min\left(\frac{\pi_j}{\delta_i}, \frac{4\pi_j}{4\delta_i+1}\right) \geq \frac{m-1}{4}$.

Lemma 3.2 *Let d and p be rational numbers, $1 \leq d < 2$ and $|p| \leq d$. Let i and j be integers, $0 \leq i < 4$ and $0 \leq j < 32$, such that $\delta_i \leq d < \delta_i + \frac{1}{4}$ and $\pi_j \leq p < \pi_j + \frac{1}{4}$. Let $m = \phi(i, j)$. Then $|4p - dm| \leq d$.*

PROOF: First note that since
$$\pi_j \leq p \leq d < \delta_i + \frac{1}{4}$$
and
$$\pi_j > p - \frac{1}{4} \geq -d - \frac{1}{4} > -\left(\delta_i + \frac{1}{4}\right) - \frac{1}{4} = -\delta_i - \frac{1}{2},$$
we may apply Lemma 3.1.

We must show that $-d \leq 4p - dm \leq d$, i.e.,
$$\frac{m-1}{4} \leq \frac{p}{d} \leq \frac{m+1}{4}.$$
First we establish the upper bound. Since
$$\frac{p}{d} \leq 1 = \frac{3+1}{4},$$

we may assume $m \neq 3$. If $\pi_j \geq -\frac{1}{4}$, then

$$\frac{p}{d} < \frac{\pi_j + \frac{1}{4}}{d} \leq \frac{\pi_j + \frac{1}{4}}{\delta_i} = \frac{4\pi_j + 1}{4\delta_i} \leq \frac{m+1}{4}.$$

On the other hand, if $\pi_j < -\frac{1}{4}$, then

$$\frac{p}{d} < \frac{\pi_j + \frac{1}{4}}{d} < \frac{\pi_j + \frac{1}{4}}{\delta_i + \frac{1}{4}} = \frac{4\pi_j + 1}{4\delta_i + 1} \leq \frac{m+1}{4}.$$

As for the lower bound, since

$$\frac{p}{d} \geq -1 = \frac{-3-1}{4},$$

we may assume $m \neq -3$. If $\pi_j \geq 0$, then

$$\frac{p}{d} \geq \frac{\pi_j}{d} \geq \frac{\pi_j}{\delta_i + \frac{1}{4}} = \frac{4\pi_j}{4\delta_i + 1} \geq \frac{m-1}{4}.$$

But if $\pi_j < 0$, then

$$\frac{p}{d} \geq \frac{\pi_j}{d} \geq \frac{\pi_j}{d} \geq \frac{m-1}{4}. \ \Box$$

4 Implementation

The results of this section refer to the values assumed by variables during a hypothetical execution of the XFL function `SRT`, defined in Appendix B. With the exception of the loop variables b and k, each variable of `SRT` belongs to one of two classes:

- Some variables assume at most one value during an execution. The value of such a variable will be denoted by the name of the variable in italics, e.g., X, *dEnc*, and *YNB*.

- Variables that are assigned inside the main `for` loop may assume only one value during each iteration and may or may not be assigned an initial value before the loop is entered. The value assigned to such a variable during the k^{th} iteration will be denoted with the subscript k, e.g., p_k, $mAbs_k$, and $addA_k$. If such a variable is assigned an initial value outside of the loop, it will be denoted with the subscript 0, e.g., p_0 and $QPart_0$. When convenient, the subscript may be omitted and understood to have the value k. When replaced with an accent ($'$), it will be understood to have the value $k-1$. For example, in the statement of Lemma 4.7, m and p' represent m_k and p_{k-1}, respectively.

`SRT` has four input parameters:

- *isSigned* is a boolean indication of a signed or unsigned integer format;

- w is the format width, which is assumed to be 8, 16, 32, or 64;

- *XEnc* is the signed or unsigned w-bit encoding of the divisor;

- *YEnc* is the signed or unsigned $2w$-bit encoding of the dividend.

Three values are returned:

- A boolean indication of whether the computation completed successfully;
- The signed or unsigned w-bit encoding of the quotient;
- The signed or unsigned w-bit encoding of the remainder.

The last two values are of interest only when the first is true, in which case they are the values of the variables $QOut$ and $ROut$, respectively.

Some of the variables of `SRT` do not contribute to the outputs, but are used only in our analysis and in embedded assertions. Of these (listed in a preamble to the function), X and Y are the integer values represented by $XEnc$ and $YEnc$, and Q and R are the quotient and remainder, which, unless $X = 0$, satisfy $Y = QX + R$, $|R| < |X|$, and either $R = 0$ or $sgn(R) = sgn(Y)$.

Our objective is to show that success is indicated if and only if $X \neq 0$ and Q is representable with respect to the indicated format, in which case Q and R are the integer values of $QOut$ and $ROut$. Since this obviously holds when $X = 0$, we shall assume $X \neq 0$ in the following. The main result is the theorem at the end of this section.

The computation is naturally partitioned into three phases, which are described in the following three subsections.

4.1 Analysis of Operands

In the first phase, the operands are analyzed and normalized in preparation for the iterative computaion of the quotient and remainder, and the number n of iterations is established.

The variable XNB represents the "number of bits" of X, derived by counting the leading zeroes or ones:

Lemma 4.1 $XNB = expo(X) + 1$.

Proof: If $X > 0$, then $X = XEnc$ and $XNB - 1$ is the index of the leading 1 of X, which implies $2^{XNB\text{-}1} \leq X < 2^{XNB}$, and the claim follows.

If $XNegPower2 = 1$, then $XEnc[b] = 1$ if and only if $w > b \geq XNB - 1$. It follows that $XEnc = 2^w - 2^{XNB-1}$ and

$$X = XEnc - 2^w = 2^w - 2^{XNB-1} - 2^w = -2^{XNB-1}.$$

In the remaining case, $X < 0$, $XNB - 1$ is the index of the leading 0 of $XEnc$, and $XEnc[XNB - 2 : 0] \neq 0$. It follows that

$$2^w - 2^{XNB} < XEnc < 2^w - 2^{XNB-1},$$

which implies $-2^{XNB} < X < -2^{XNB-1}$, i.e., $2^{XNB-1} < |X| < 2^{XNB}$. □

The variable $dEnc$ is an encoding of $d = 2^{-expo(X)}X$:

Lemma 4.2 $dEnc = (2^{65}d)[67 : 0] = d[2 : -65]$.

PROOF: Since $d = 2^{-expo(X)}X$ and $expo(X) \leq 63$, $2^{63}d = 2^{63-expo(X)}X$ is an integer and

$$X = 2^{expo(X)}d = 2^{XNB-1}d.$$

Clearly,

$$dEnc[65:66-XNB] = XEnc[XNB-1:0] = X[w-1:0][XNB-1:0] = X[XNB-1:0]$$

and since $|X| < 2^{XNB}$,

$$dEnc[66] = dEnc[67] = XSign = X[XNB] = X[XNB+1].$$

Thus, $dEnc[67:66-XNB] = X[XNB+1:0]$, and hence

$$\begin{aligned} dEnc &= 2^{66-XNB}X[XNB+1:0] \\ &= (2^{66-XNB}X)[67:0] \\ &= (2^{XNB-1+66-XNB}d)[67:0] \\ &= (2^{65}d)[67:0] \\ &= d[2:-65]. \ \Box \end{aligned}$$

The next lemma gives an expression for i, the first argument of the table access function ϕ:

Lemma 4.3 $i = \lfloor 4(|d|-1) \rfloor$.

PROOF: If $X > 0$, then since $4 \leq 4d < 8$,

$$i = dEnc[64:63] = d[-1:-2] = mod(\lfloor 4d \rfloor, 4) = \lfloor 4d \rfloor - 4 = \lfloor 4(d-1) \rfloor.$$

If $XNegPower2 = 1$, then $X = -2^{expo(X)}$, $d = -1$, and

$$i = 0 = \lfloor 4(|d|-1) \rfloor.$$

In the remaining case, $X < 0$, $dEnc[66] = 1$, $dEnc[65] = 0$, and $dEnc[64:0] \neq 0$. Since $2^{65}d = 2^{65-expo(X)}X$ is an integer and $2^{65}d < 2^{66}$,

$$\begin{aligned} 2^{65}d &= SgndIntVal(67, (2^{65}d)[66:0]) \\ &= SgndIntVal(67, dEnc[66:0]) \\ &= dEnc[64:0] - 2^{66}. \end{aligned}$$

Thus, $|d| = -d = 2 - 2^{-65}dEnc[64:0]$ and

$$\lfloor 4(|d|-1) \rfloor = \lfloor 4 - 2^{-63}dEnc[64:0] \rfloor = \lfloor 4 - dEnc[64:63] - 2^{-63}dEnc[62:0] \rfloor.$$

Suppose that $dEnc[62:0] = 0$. Then $dEnc[64:63] \neq 0$, and

$$\lfloor 4(|d|-1) \rfloor = 4 - dEnc[64:63].$$

An exhaustive case analysis ($dEnc[64:63] = 1$, 2, or 3) shows that

$$4 - dEnc[64:63] = ((\text{~}dEnc[64] \mid \text{~}dEnc[63]) << 1) \mid dEnc[63] = i.$$

Finally, suppose that $dEnc[62:0] \neq 0$. Then

$$\lfloor 4(|d|-1) \rfloor = 3 - dEnc[64:63] = \tilde{}dEnc[64:63] = i. \ \Box$$

YNB is the "number of bits" of Y, including, in the negative case, the final trailing sign bit:

Lemma 4.4 *If $Y > 0$, then*

$$2^{YNB-1} \leq Y < 2^{YNB},$$

and if $Y < 0$, then

$$2^{YNB-2} < |Y| \leq 2^{YNB-1}.$$

Consequently, in either case, $YNB \geq expo(Y) + 1$.

PROOF: If $Y > 0$, then $YNB - 1$ is the index of the leading 1 of $YEnc = Y$, i.e., $expo(Y) = $ *YNB-1.*

If $Y = -1$, then $YNB = 1$ and

$$2^{YNB-2} = \frac{1}{2} < |Y| = 1 = 2^{YNB-1}.$$

In the remaining case, $Y < -1$, $YNB - 2$ is the index of the leading 0 of *YEnc*, which implies

$$2^w - 2^{YNB-1} \leq YEnc < 2^w - 2^{YNB-2}.$$

But since $Y = YEnc - 2^w$,

$$-2^{YNB-1} \leq Y < -2^{YNB-2}$$

and

$$2^{YNB-2} < |Y| \leq 2^{YNB-1}. \ \Box$$

The number of iterations, n, satisfies the requirement of Lemma 2.1:

Lemma 4.5 $|Y| < 2^{2n}|X|$.

PROOF: This is an immediate consequence of Lemmas 4.1, 4.4, and 2.5. $\Box$

The bit vector *pEnc* is an encoding of $p_0 = 2^{-expo(X)-2n}Y$:

Lemma 4.6

(a) If $n = 0$, then

$$pEncHi_0 = \begin{cases} (2^{64-YNB}Y)[67:0] & \text{if } YNB[0] = XNB[0] \\ (2^{65-YNB}Y)[67:0] & \text{if } YNB[0] \neq XNB[0]. \end{cases}$$

(b) If $n > 0$, then $2^{129}p_0$ is an integer and

$$pEnc = (2^{129}p_0)[131:0] = p_0[2:-129].$$

PROOF: First consider the case $\mathit{YNB}[0] = \mathit{XNB}[0]$. We may assume $\mathit{YNB} > 0$; otherwise, $Y = 0$ and the lemma is trivial.

Note that $\mathit{YNB} \leq 128$ and

$$\mathit{pEnc}[127 : 128 - \mathit{YNB}] = \mathit{YEnc}[\mathit{YNB} - 1 : 0] = Y[\mathit{YNB} - 1 : 0].$$

Therefore,

$$\mathit{pEnc}[127 : 0] = 2^{128-\mathit{YNB}}Y[\mathit{YNB} - 1 : 0] = (2^{128-\mathit{YNB}}Y)[127 : 0].$$

Since $Y < 2^{\mathit{YNB}}$, for $\ell = 128, \ldots, 131$,

$$\mathit{pEnc}[\ell] = \mathit{YSign} = (2^{128-\mathit{YNB}}Y)[\ell].$$

Thus,

$$\mathit{pEnc} = \mathit{pEnc}[131 : 0] = (2^{128-\mathit{YNB}}Y)[131 : 0].$$

If $n = 0$, then $\mathit{YNB} < \mathit{XNB} \leq 64$ and

$$\mathit{pEnc} = (2^{128-\mathit{YNB}}Y)[131 : 0] = 2^{64}(2^{64-\mathit{YNB}}Y)[67 : 0],$$

which implies $\mathit{pEncHi}_0 = (2^{64-\mathit{YNB}}Y)[67 : 0]$.

On the other hand, if $n > 0$, then

$$2n = 2\left(\left\lfloor \frac{\mathit{YNB} - \mathit{XNB}}{2} \right\rfloor + 1\right) = \mathit{YNB} - \mathit{XNB} + 2 = \mathit{YNB} - \mathit{expo}(X) + 1$$

and

$$p_0 = 2^{-\mathit{expo}(X)-2n}Y = 2^{-\mathit{YNB}-1}Y.$$

Thus, $2^{129}p_0 = 2^{128-\mathit{YNB}}Y$ is an integer and

$$\mathit{pEnc} = (2^{128-\mathit{YNB}}Y)[131 : 0] = (2^{129}p_0)[131 : 0].$$

The proof for the case $\mathit{YNB}[0] \neq \mathit{XNB}[0]$ is similar, with every occurrence of 127 or 128 replaced by 128 or 129. Thus, we have

$$\mathit{pEnc} = (2^{129-\mathit{YNB}}Y)[131 : 0],$$

which, in the $n = 0$ case, leads to

$$\mathit{pEncHi}_0 = (2^{65-\mathit{YNB}}Y)[67 : 0]. \ \square$$

4.2 Iteration

The second phase is the iteration loop, in which the quotient digits are selected and the partial remainder and quotient are updated accordingly. The main results pertaining to the iterative computation of the partial remainder are given by Lemmas 4.7 and 4.9:

(1) The quotient digit m is correctly computed as the value of $\phi(i, j)$, as stated in Lemma 4.7;

(2) The partial remainder $p_k = 4p_{k-1} - m_k d$ is encoded by *pEncHi*, *carryHi*, and *pEncLo*, as stated in Lemma 4.9.

The proof of (2) depends on (1), and that of (1) requires the assumption that (2) holds on the preceding iteration:

Lemma 4.7 *Let $0 < k \leq n$. Suppose that $|p'| \leq |d| < 2$, $2^{129}p'$ is an integer, and*

$$(2^{129}p')[131:0] = 2^{64}(\mathit{pEncHi}' + \mathit{carryHi}')[67:0] + \mathit{pEncLo}'.$$

Then

(a) $m = \begin{cases} \phi(i,j) & \text{if } X \geq 0 \\ -\phi(i,j) & \text{if } X < 0; \end{cases}$

(b) $\pi_j \leq p' < \pi_j + \frac{1}{4}$.

Proof: First suppose $\mathit{pEncHi}' + \mathit{carryHi}' \geq 2^{68}$. Then $\mathit{pTop} = 63$; otherwise,

$$\begin{aligned} \mathit{pEncHi}' + \mathit{carryHi}' &= 2^{62}\mathit{pTop} + \mathit{pEncHi}'[61:0] + \mathit{carryHi}' \\ &\leq 2^{62}62 + 2^{62} - 1 + 2^{62} - 1 \\ &< 2^{68}. \end{aligned}$$

Consequently, $j = \mathit{pIndex} = \mathit{pTop}[4:0] = 31$, which implies $m = \phi(i,j) = 0$, and (a) follows. To prove (b), we note that

$$\begin{aligned} (2^{129}p')[131:64] &= (\mathit{pEncHi}' + \mathit{carryHi}')[67:0] \\ &= \mathit{pEncHi}' + \mathit{carryHi}' - 2^{68} \\ &< 2^{68} + 2^{62} - 2^{68} \\ &= 2^{62} \end{aligned}$$

and therefore,

$$(2^{129}p')[131:0] < 2^{64}(2^{62} - 1) + 2^{62} < 2^{126}.$$

Since $|2^{129}p'| < 2^{131}$,

$$2^{129}p' = \mathit{SgndIntVal}(132, (2^{129}p')[131:0]) = (2^{129}p')[131:0]$$

and thus, $0 \leq 2^{129}p' < 2^{126}$ and

$$\pi_j = -\frac{1}{8} < 0 \leq p' < \frac{1}{8} = \pi_j + \frac{1}{4}.$$

We may assume, therefore, that $\mathit{pEncHi}' + \mathit{carryHi}' < 2^{68}$ and hence

$$\begin{aligned} (2^{129}p')[131:0] &= 2^{64}(\mathit{pEncHi}' + \mathit{carryHi}')[67:0] + \mathit{pEncLo}' \\ &= 2^{64}(\mathit{pEncHi}' + \mathit{carryHi}') + \mathit{pEncLo}' \\ &= 2^{126}\mathit{pTop} + 2^{64}(\mathit{pEncHi}'[61:0] + \mathit{carryHi}') + \mathit{pEncLo}' \\ &< 2^{126}\mathit{pTop} + 2^{64}(2^{62} - 1 + 2^{62} - 1) + 2^{64} \\ &< 2^{126}(\mathit{pTop} + 2). \end{aligned}$$

Suppose $p' \geq 0$. Then $2^{129}p' = (2^{129}p')[131:0]$ and

$$\frac{1}{8}\mathit{pTop} \leq p' < \frac{1}{8}\mathit{pTop} + \frac{1}{4}.$$

Since $pTop \leq 8p' < 16$, $j = pIndex = pTop$ and $pSign = 0$. Thus,

$$|m| = mAbs = SRTLookup(i,j) = |\phi(i,j)| = \phi(i,j)$$

and $mSign = XSign$, which implies (a). To prove (b), we need only observe that

$$SgndIntVal(5,j) = SgndIntVal(5, pTop) = pTop.$$

Now suppose $p' < 0$. Then $2^{129}p' = (2^{129}p')[131:0] - 2^{132}$ and the above estimate yields

$$\frac{1}{8}(pTop - 64) \leq p' < \frac{1}{8}(pTop - 64) + \frac{1}{4}.$$

Thus, $pTop > 8p' + 62 > -16 + 62 = 46$, so $pTop \geq 47$. Let us assume that $pTop \geq 48$. Then $j = pIndex$ and $|m| = |\phi(i,j)| = -\phi(i,j)$. Thus, to establish (a), we need only show that m and X have opposite signs. But this follows from $mSign = XSign \mathbin{\hat{}} pSign$ and $pSign = 1$. To prove (b), it suffices to show that $pTop = SgndIntVal(5,j) + 64$. But in this case, $j = pTop[4:0] = pTop - 32 \geq 16$, so $SgndIntVal(5,j) = j - 32 = pTop - 64$.

There remains the special case $pTop = 47$. Since

$$p' < \frac{1}{8}(pTop - 64) + \frac{1}{4} = -\frac{17}{8} + +\frac{1}{4} = -\frac{15}{8},$$

$2 > |d| \geq |p'| > \frac{15}{8}$, which implies

$$dIndex = i = \lfloor 4(|d| - 1) \rfloor = 3.$$

Thus,

$$|m| = mAbs = SRTLookup(3, 15) = 3.$$

On the other hand, $\phi(i,j) = \phi(3,16) = -3$. But again, since $pSign = 1$, m and X have opposite signs and (a) follows. To prove (b), note that $SgndIntVal(5,j) = -16$; hence,

$$\pi_j = -2 < p' < -\frac{17}{8} + \frac{1}{4} < \pi_j + \frac{1}{4}. \ \Box$$

The computation of the partial remainder, as described in Lemma 4.9, involves a "compression" that reduces four addends to two. This is performed by the serial operation of two carry-save adders, as described by the following basic result, taken from [10]:

Lemma 4.8 *Given n-bit vectors x, y, and z, let*

$$a = x \mathbin{\hat{}} y \mathbin{\hat{}} z$$

and

$$b = 2(x \,\&\, y \mid x \,\&\, z \mid y \,\&\, z).$$

Then

$$x + y + z = a + b.$$

Lemma 4.9 *If $n > 0$ and $0 \leq k \leq n$, then $|p| \leq |d| < 2$, $2^{129}p$ is an integer, and*

$$(2^{129}p)[131:0] = p[2:-129] = 2^{64}(pEncHi + carryHi)[67:0] + pEncLo.$$

PROOF: The proof is by induction on k.

For $k = 0$, we have

$$|p| = |2^{-expo(X)-2n}Y| < |2^{-expo(X)}X| = |d|$$

by Lemma 4.5, and since

$$2^{64}(pEncHi + carryHi)[67:0] + pEncLo = pEnc,$$

the other two claims follow from Lemma 4.6.

In the inductive case, we shall derive the bound on $|p|$ from Lemma 3.2. By Lemma 4.3, since $|d| \leq 1$, $0 \leq i < 4$ and $\delta_i \leq |d| < \delta_i + \frac{1}{4}$. Clearly, $j < 32$, and by Lemma 4.6, $\pi_j \leq p < \pi_j + \frac{1}{4}$ and

$$m = \begin{cases} \phi(i,j) & \text{if } X \geq 0 \\ -\phi(i,j) & \text{if } X < 0. \end{cases}$$

Thus, applying Lemma 3.2, with the signs of d and m reversed if $d < 0$, we have $|p| = |4p' - md| \leq d$.

By induction, $2^{129}p = 2^{131}p' - 2^{129}md$ is an integer. The computation of $(2^{129}p)[131:0]$ involves a 4–2 compressor with inputs *addA*, *addB*, *addC*, *addD*. We shall show that

$$(addA + addB + addC + addD)[67:0] = (2^{129}p)[131:64].$$

The first two terms, *addA* and *addB*, if not 0, represent $\pm 2d$ and $\pm d$, respectively, depending on the value of m. However, in the negative case, in order to avoid a full 67-bit addition, the simple complement of $2d$ or d is used in place of its negation, and the missing 1 is recorded in the variable *inject*, which is more conveniently combined later with *addD*. Thus, our first goal is to prove that

$$(addA + addB + inject)[67:0] = (-2^{65}dm)[67:0].$$

If $mSign = 1$, then

$$addA = mAbs[1] \cdot (2 \cdot dEnc)[67:0] = (2 \cdot mAbs[1] \cdot dEnc)[67:0],$$

$$addB = mAbs[0] \cdot dEnc,$$

and $inject = 0$. Hence,

$$\begin{aligned}(addA + addB + inject)[67:0] &= (2 \cdot mAbs[1] \cdot dEnc + mAbs[0] \cdot dEnc)[67:0] \\ &= ((2 \cdot mAbs[1] + mAbs[0]) \cdot dEnc)[67:0] \\ &= (mAbs \cdot dEnc)[67:0] \\ &= (-dEnc \cdot m)[67:0]) \\ &= (-2^{65}dm)[67:0]).\end{aligned}$$

On the other hand, if $mSign = 0$, then

$$\begin{aligned}addA &= addA[67:0] \\ &= mAbs[1] \cdot (2(\tilde{}\,dEnc[66:0]) + 1)[67:0] \\ &= mAbs[1] \cdot (2(-dEnc - 1)[66:0]) + 1)[67:0] \\ &= mAbs[1] \cdot (-2dEnc - 2)[67:0]) + 1)[67:0] \\ &= mAbs[1] \cdot (-2 \cdot dEnc - 1)[67:0] \\ &= (-2 \cdot mAbs[1] \cdot dEnc - mAbs[1])[67:0],\end{aligned}$$

$$\begin{aligned} addB &= mAbs[0] \cdot \tilde{\ } dEnc[67:0] \\ &= mAbs[0] \cdot (-dEnc - 1)[67:0] \\ &= (-mAbs[0] \cdot dEnc - mAbs[0])[67:0], \end{aligned}$$

and $inject = mAbs[0] + mAbs[1]$, so that

$$\begin{aligned} &(addA + addB + inject)[67:0] \\ &\quad = (-2 \cdot mAbs[1] \cdot dEnc - mAbs[1] - mAbs[0] \cdot dEnc - mAbs[0] + mAbs[0] + mAbs[1])[67:0] \\ &\quad = (-(2 \cdot mAbs[1] + mAbs[0]) \cdot dEnc)[67:0] \\ &\quad = (-m \cdot dEnc)[67:0] \\ &\quad = (-2^{65}dm)[67:0]) \\ &\quad = (-2^{129}dm)[131:64]). \end{aligned}$$

The remaining two terms, *addC* and *addD*, represent the shifted result of the previous iteration, $4p'$. Thus,

$$addC = 4 \cdot pEncHi + pEncLo[63:62],$$

$$addD = 2 \cdot carryHi + inject,$$

and

$$\begin{aligned} &(addC + addD - \text{inject})[67:0] \\ &\quad = (4 \cdot pEncHi + pEncLo[63:62] + 4 \cdot carryHi + \text{inject} - \text{inject})[67:0] \\ &\quad = (4(pEncHi + carryHi)[67:0] + pEncLo[63:62])[67:0] \\ &\quad = (4(2^{64}(pEncHi + carryHi)[67:0] + 2^{62} \cdot pEncLo[63:62]))[131:64] \\ &\quad = 4(2^{64}(pEncHi + carryHi)[67:0] + 2^{62} \cdot pEncLo[63:62])[129:62] \\ &\quad = 4(2^{64}(pEncHi + carryHi)[67:0] + 2^{62} \cdot pEncLo[63:62] + pEncLo[61:0])[129:62] \\ &\quad = 4(2^{64}(pEncHi + carryHi)[67:0] + pEncLo)[129:62] \\ &\quad = 4(2^{129}p')[131:0][129:62] \\ &\quad = 4(2^{129}p')[129:62] \\ &\quad = (4 \cdot 2^{129}p')[131:64]. \end{aligned}$$

Combining these last two results, we have

$$\begin{aligned} &(addA + addB + addC + addD)[67:0] \\ &\quad = ((addA + addB + inject)[67:0] + (addC + addD - inject)[67:0])[67:0] \\ &\quad = ((2^{129}4p')[131:64] + (-2^{129}dm)[131:64])[67:0]. \end{aligned}$$

Since $(-2^{129}dm)[63:0] = 0$, this may be reduced to

$$((2^{129}(4p' - dm))[131:64] = (2^{129}p)[131:64].$$

Two applications of Lemma 4.8 yield

$$addA + addB + addC + addD = sum1 + carry1 + addD = sum2 + carry2,$$

and therefore,

$$\begin{array}{rl}(2^{129}p)[131:64] & \\ = & (\mathit{sum2}+\mathit{carry2})[67:0] \\ = & ((2^{62}(\mathit{sum2}[67:62]+\mathit{carry2}[67:62]))[67:0]+\mathit{sum2}[61:0]+\mathit{carry2}[61:0])[67:0] \\ = & (2^{62}(\mathit{sum2}[67:62]+\mathit{carry2}[67:62])[5:0]+\mathit{sum2}[61:0]+\mathit{carry2}[61:0])[67:0] \\ = & (\mathit{pEncHi}+\mathit{carryHi})[67:0].\end{array}$$

But by Lemma 4.2,

$$\begin{array}{rcl}(2^{129}p)[63:0] & = & (2^{129}(4p'-md))[63:0] \\ & = & ((2^{129}4p')[63:0]+(-2^{64}m2^{65}d)[63:0])[63:0] \\ & = & (2^{129}4p')[63:0] \\ & = & (4\cdot \mathit{pEncLo}')[63:0] \\ & = & \mathit{pEncLo},\end{array}$$

and thus,

$$\begin{array}{rcl}(2^{129}p)[131:0] & = & 2^{64}(2^{129}p)[131:64]+(2^{129}p)[63:0] \\ & = & 2^{64}(\mathit{pEncHi}+\mathit{carryHi})[67:0]+\mathit{pEncLo}.\ \Box\end{array}$$

As a result of the iterative shifting of the partial remainder, $\mathit{pEncLo}=0$ upon exiting the loop. This is proved recursively:

Lemma 4.10 *If $n>0$ and $0\leq k\leq n$, then $\mathit{pEncLo}[63-2(n-k):0]=0$).*

PROOF: The proof is by induction on k. For $k=0$, since $\mathit{pEncLo}[127-YNB:0]=0$, we need only show that $127-\mathit{YNB}\geq 63-2n$, or $2n\geq \mathit{YNB}-64$. But

$$2n=2\left(\left\lfloor\frac{\mathit{YNB}-\mathit{XNB}}{2}\right\rfloor+1\right)\geq \mathit{YNB}-\mathit{XNB}+1\geq \mathit{YNB}-63.$$

For $k>0$,

$$\begin{array}{rcl}\mathit{pEncLo}[63-2(n-k):0] & = & (4\cdot \mathit{pEncLo}')[63-2(n-k):0] \\ & = & 4\cdot \mathit{pEncLo}'[63-2(n-k+1):0] \\ & = & 0.\ \Box\end{array}$$

The partial quotient *QPart* is encoded by *Q0Enc*. Its computation, as described in Lemma 2.4, is facilitated by simultaneously maintaining encodings of $\mathit{QPart}\pm 1$:

Lemma 4.11 *For $0\leq k\leq n$,*

$$\mathit{Q0Enc}=\mathit{QPart}[66:0],$$

$$\mathit{QMEnc}=(\mathit{QPart}-1)[66:0],$$

and

$$\mathit{QPEnc}=(\mathit{QPart}+1)[66:0].$$

PROOF: We shall invoke Lemma 2.4 with $N = 67$ and $Q_k = QPart$. We need only show that $\mathit{Q0Enc} = E_k$, $\mathit{QMEnc} = E_k^-$, and $\mathit{QPEnc} = E_k^+$. The claim is trivial for $k = 0$. For $k > 0$, it may be readily verified by examining each value of m, $-3 \leq m \leq 3$. For example, if $m = -1$, then $\mathit{mSign} = 1$, $\mathit{mAbs} = 1$,

$$\mathit{Q0Enc} = (4 \cdot \mathit{Q0Enc}')[66:0] = (4E_{k-1})[66:0] = (4E_{k-1} + m_k - 1)[66:0] = E_k^+,$$

$$\begin{aligned} \mathit{QMEnc} &= (4 \cdot \mathit{Q0Enc}')[66:0] \mid 2 \\ &= (4E_{k-1}^-)[66:0] \mid 2 \\ &= (4E_{k-1}^- \mid 2)[66:0] \\ &= (4E_{k-1}^- + 2)[66:0] \\ &= (4E_{k-1}^- + m_k + 3)[66:0] \\ &= E_k^-, \end{aligned}$$

and

$$\begin{aligned} \mathit{QPEnc} &= (4 \cdot \mathit{Q0Enc}')[66:0] \mid 3 \\ &= (4E_{k-1}^-)[66:0] \mid 3 \\ &= (4E_{k-1}^- \mid 3)[66:0] \\ &= (4E_{k-1}^- + 3)[66:0] \\ &= (4E_{k-1}^- + m_k + 4)[66:0] \\ &= E_k^+. \ \Box \end{aligned}$$

4.3 Final Computation

In the final phase of the computation, a full addition is performed to generate an explicit (non-redundant) representation of the remainder. This result is then adjusted, along with the quotient, to produce the final results as specified by Lemma 4.18.

The next lemma refers to the quotient and remainder before the correction step:

Lemma 4.12 *$Y = \mathit{QPre} \cdot X + \mathit{RPre}$ and $|\mathit{RPre}| \leq |X|$.*

PROOF : This is an immediate consequence of Lemma 2.1, with $r = 2$, $Q_k = \mathit{QPart}_k$, $R = \mathit{RPre}$, and $Q = \mathit{QPre}$. We need only note that the condition $|p_k| \leq |d|$ is ensured by Lemma 4.9. $\Box$

RPre is encoded by *REncPre*:

Lemma 4.13

$$\mathit{REncPre} = \begin{cases} (2^{66-\mathit{XNB}}\mathit{RPre})[66:0] & \text{if } n > 0 \\ (2^{64-\mathit{YNB}}\mathit{RPre})[66:0] & \text{if } n = 0 \text{ and } \mathit{YNB}[0] = \mathit{XNB}[0] \\ (2^{65-\mathit{YNB}}\mathit{RPre})[66:0] & \text{if } n = 0 \text{ and } \mathit{YNB}[0] \neq \mathit{XNB}[0]. \end{cases}$$

Proof: If $n > 0$, then by Lemmas 4.9 and 4.1,

$$\begin{aligned} REncPre &= (pEncHi_n + carryHi_n)[66:0] \\ &= (2^{129}p_n)[130:64] \\ &= (2^{64}2^{65-expo(X)}RPre)[130:64] \\ &= (2^{66-expo(X)}RPre)[66:0] \\ &= (2^{66-XNB}RPre)[66:0]. \end{aligned}$$

On the other hand, if $n = 0$, then $REncPre = pEncHi_0[66:0]$ and the lemma follows from Lemma 4.6. □

The encoding *REnc* of the final remainder, which is derived from *REncPre*, depends on the signs of *RPre* and Y and the special cases *RPre* is 0 or $\pm X$. Timing considerations dictate that these conditions must be detected before the full addition that produces *REncPre* is actually performed. This requires a technique for predicting cancellation, which is provided by the following result, found in [10]:

Lemma 4.14 *Given n-bit vectors a and b and a one-bit vector c, let*

$$\tau = a \mathbin{\hat{}} b \mathbin{\hat{}} (2(a \mid b) + c).$$

If $0 \leq k < n$, *then*

$$(a + b + c)[k:0] = 0 \Leftrightarrow \tau[k:0] = 0.$$

Lemma 4.14 is used in the proofs of the following three lemmas:

Lemma 4.15 *RIs0 is true if and only if RPre* $= 0$.

Proof: By Lemma 4.14, *RIs0* is true if and only if $REncPre = 0$. If $n > 0$, then by Lemma 4.13, $REncPre = (2^{65-expo(X)}RPre)[66:0]$. But by Lemma 4.12,

$$|2^{65-expo(X)}RPre| \leq |2^{65-expo(X)}X| < 2^{66},$$

and it follows that $REncPre = 0$ if and only if $RPre = 0$.

Now suppose $n = 0$. Then $RPre = 2^{expo(X)}p_0 = Y$, and $|Y| < 2^{YNB} \leq 2^{XNB-1} \leq |X|$. By Lemma 4.13, $REncPre = (2^{e-YNB}RPre)[66:0]$, where $e = 64$ or 65. Thus,

$$|2^{e-YNB}RPre| \leq |2^{65-YNB}Y| \leq |2^{64-expo(Y)}Y| < 2^{65},$$

and again, $REncPre = 0$ if and only if $RPre = 0$. □

Lemma 4.16 *RNegX is true if and only if RPre* $= -X$.

Proof: First note that by Lemma 4.2,

$$dEnc[66:0] = (2^{65}d)[66:0] = (2^{65-expo(X)}X)[66:0].$$

Now by Lemma 4.8,

$$\begin{aligned} (RNegXSum + RNegXCarry)[66:0] &= (pEncHi + CarryHi + dEnc)[66:0] \\ &= (REncPre + dEnc)[66:0]; \end{aligned}$$

hence, by Lemma 4.14,

$$\begin{aligned} RNegX = 1 &\Leftrightarrow (RNegXSum + RNegXCarry)[66:0] = 0 \\ &\Leftrightarrow (REncPre + dEnc))[66:0] = 0 \\ &\Leftrightarrow (REncPre + 2^{65-expo(X)}X)[66:0] = 0. \end{aligned}$$

If $n > 0$, then we have

$$RNegX = 1 \Leftrightarrow (2^{65-expo(X)}(RPre + X))[66:0] = 0,$$

where

$$|2^{65-expo(X)}(RPre + X)| < |2^{65-expo(X)}(2X)| \leq |2^{65-expo(X)}2^{expo(X)+2}| = 2^{67},$$

and the result follows.

If $n = 0$, then since $|RPre| = |Y| < |X|$, we must show that $RNegX = 0$. By Lemma 4.15, $REncPre = (2^{e-YNB}Y)[66:0]$, where $e \leq 65$ and as noted above, $|2^{e-YNB}Y| < 2^{65}$. If $RNegX = 1$, then $(2^{e-YNB}Y + 2^{65-expo(X)}X)[66:0] = 0$. But since

$$|2^{e-YNB}Y + 2^{65-expo(X)}X| < 2|2^{65-expo(X)}X| < 2|2^{65-expo(X)}2^{expo(X)+1}| = 2^{67},$$

this implies $2^{e-YNB}Y + 2^{65-expo(X)}X = 0$, which is impossible. □

Lemma 4.17 *RPosX is true if and only if $RPre = X$.*

Proof: By Lemma 4.8,

$$\begin{aligned} (RPosXSum + RPosXCarry)[66:0] &= (pEncHi + CarryHi + \tilde{}dEnc + 1)[66:0] \\ &= (REncPre - dEnc - 1 + 1)[66:0] \\ &= (REncPre - dEnc)[66:0], \end{aligned}$$

and hence, by Lemma 4.14,

$$\begin{aligned} RPosX = 1 &\Leftrightarrow (RPosXSum + RPosXCarry)[66:0] = 0 \\ &\Leftrightarrow (REncPre - dEnc)[66:0] = 0. \end{aligned}$$

The rest of the proof is similar to that of Lemma 4.16. □

Lemma 4.18 *$Y = QX + R$, where $|R| < |X|$ and either $R = 0$ or $sgn(R) = sgn(Y)$.*

Proof: This is an immediate consequence of Lemmas 2.2, 4.12, and 4.15. □

The final remainder is encoded by *REnc*:

Lemma 4.19

$$REnc = \begin{cases} (2^{64-XNB}R)[63:0] & \textit{if } n > 0 \\ (2^{62-YNB}R)[63:0] & \textit{if } n = 0 \textit{ and } YNB[0] = XNB[0] \\ (2^{63-YNB}R)[63:0] & \textit{if } n = 0 \textit{ and } YNB[0] \neq XNB[0]. \end{cases}$$

PROOF: If *fixupNeeded* is false, then $R = RPre$, $REnc = REncPre[66:2]$, and the lemma follows from Lemma 4.13. If $n = 0$, then as noted in the proof of Lemma 4.15, $RPre = Y$ and $|Y| < |X|$, from which it follows that *fixupNeeded* is false. Thus, we may assume that *fixupNeeded* is true and $n > 0$. We may further assume that $RIsX = 0$; otherwise, $REnc = R = 0$. If $RSign = XSign$, then

$$\begin{aligned} REnc &= (REncPre + \tilde{\ } dEnc[66:0] + 1)[65:2] \\ &= (2^{66-XNB} RPre + 2^{67} - 2^{65-expo(X)} X - 1 + 1)[65:2] \\ &= (2^{66-XNB}(RPre - X))[65:2] \\ &= (2^{66-XNB} R)[65:2] \\ &= (2^{64-XNB} R)[63:0]. \end{aligned}$$

The case $RSign \neq XSign$ is similar. □

Our main result follows:

Theorem 1 *If Q is representable in the integer format determined by isSigned and w, then $QTooLarge = false$ and $QOut$ and $ROut$ are the encodings of Q and R, respectively. Otherwise, $QTooLarge$ = true.*

PROOF: We shall first prove that $QTooLarge$ is false if and only if Q is representable. We begin with the case $YNB - XNB > w$, in which must show that Q is not representable. If $Y > 0$, then by Lemmas 4.1 and 4.4, $|X| < 2^{XNB}$ and $Y \geq 2^{YNB-1}$, and hence,

$$\left|\frac{Y}{X}\right| > 2^{YNB-1-XNB} \geq 2^w,$$

which implies

$$|Q| = \left|\frac{Y-R}{X}\right| \geq \left|\frac{Y}{X}\right| - \left|\frac{R}{X}\right| > \left|\frac{Y}{X}\right| - 1 > 2^w - 1$$

and $|Q| \geq 2^w$.

Now suppose $Y < 0$. Then $|X| < 2^{XNB}$and $|Y| > 2^{YNB-2}$. Since the format is signed, it will suffice to show that $|Q| > 2^{w-1}$, or $\left|\frac{Y}{X}\right| \geq 2^{w-1} + 1$. If $XNB = w$, then $|X| \geq 2^{w-1}$ and we must have $X = -2^{w-1}$ and

$$\left|\frac{Y}{X}\right| > \frac{2^{YNB-2}}{2^{XNB-1}} = 2^{YNB-XNB-1} \geq 2^w.$$

We may assume, therefore, that $XNB < w$. Since $|X| \leq 2^{XNB} - 1$,

$$\left|\frac{Y}{X}\right| > \frac{2^{YNB-2}}{2^{XNB}-1} \geq \frac{2^{XNB+w-1}}{2^{XNB}-1},$$

and we need only show that

$$\frac{2^{XNB+w-1}}{2^{XNB}-1} \geq 2^{w-1} + 1,$$

or equivalently,

$$2^{XNB+w-1} \geq (2^{XNB} - 1)(2^{w-1} + 1) = 2^{XNB+w-1} + 2^{XNB} - 2^{w-1} - 1,$$

which follows from $XNB < w$.

In the case $n \leq 1$, $QTooLarge$ is false and we must show that Q is representable. But this is trivially true, since $YNB - XNB \leq 1$, $|Y| < 2^{YNB}$, and $X \geq 2^{XNB-1}$ imply

$$|Q| \leq \left|\frac{Y}{X}\right| < 2^{YNB-XNB+1} \leq 4.$$

In the remaining case, $YNB - XNB \leq w$ and $n > 1$. The first of these conditions implies that

$$2n \leq 2\left(\left\lfloor \frac{w}{2} \right\rfloor + 1\right) \leq w + 2;$$

thus, by Lemma 2.3,

$$|QPre| = |QPart_n| < 2^{w+2},$$

from which we conclude that $|Q| \leq 2^{w+2}$.

The second condition implies that $YNB - XNB \geq 2$, so

$$|Y| > 2^{YNB-2} \geq 2^{XNB} > |X|,$$

from which we conclude that $Q \neq 0$. Thus, if $YSign = XSign$, then $Q > 0$, and if $YSign \neq XSign$, then $Q < 0$.

Suppose $Q > 0$. Then since $Q \leq 2^{w+2}$, $Q = Q[w+2:0] = QEnc[w+2:0]$. If the format is unsigned, then

$$Q \text{ is representable} \Leftrightarrow Q < 2^w \Leftrightarrow QEnc[w+2:w] = 0 \Leftrightarrow QTooLarge = 0,$$

while if the format is signed, then

$$Q \text{ is representable} \Leftrightarrow Q < 2^{w-1} \Leftrightarrow QEnc[w+2:w-1] = 0 \Leftrightarrow QTooLarge = 0.$$

Finally, suppose $Q < 0$. Then $Q \geq -2^{w+2}$ and

$$QEnc[w+1:0] = Q[w+1:0] = mod(Q, 2^{w+2}) = Q + 2^{w+2}.$$

Since the format must be signed,

$$\begin{aligned} Q \text{ is representable} &\Leftrightarrow Q \geq -2^{w-1} \\ &\Leftrightarrow QEnc[w+1:0] \geq 2^{w+2} - 2^{w-1} \\ &\Leftrightarrow QEnc[w+1:w-1] = 7 \\ &\Leftrightarrow QTooLarge = 0. \ \Box \end{aligned}$$

Next, we show that if Q is representable, then $QOut$ and $ROut$ are the encodings of Q and R. Clearly, $QOut = QEnc[w-1:0] = Q[w-1:0]$, which is the encoding of Q. We must also show that $ROut = R[w-1:0]$.

Consider the case $n > 0$. By Lemma 4.19,

$$REnc[63:64-XNB] = (2^{64-XNB}R)[63:0][63:64-XNB] = R[XNB-1:0].$$

Since $|R| < |X| < 2^{XNB}$,

$$R[w-1:XNB] = \begin{cases} 0 & \text{if } R \geq 0 \\ 2^{w-XNB} - 1 & \text{if } R < 0 \end{cases}$$

and in either case, $R[w-1:XNB] = ROut[w-1:XNB]$.

Suppose $n = 0$ and $YNB[0] = XNB[0]$. By Lemma 4.19,

$$\begin{aligned} REnc[63:62-YNB] &= (2^{62-YNB}R)[63:0][63:62-YNB] \\ &= (2^{62-YNB}R)[63:62-YNB] \\ &= R[YNB+1:0]. \end{aligned}$$

Since $|R| = |Y| < 2^{YNB}$,

$$R[w-1:YNB+1] = \begin{cases} 0 & \text{if } R \geq 0 \\ 2^{w-YNB-1}-1 & \text{if } R < 0 \end{cases}$$

and in either case, $R[w-1:YNB+1] = ROut[w-1:YNB+1]$.

The case $YNB[0] \neq XNB[0]$ is similar. $\square$

Acknowledgment

The author is grateful to Mike Achenbach, the principal designer of the Llano divider, for facilitating this work, and especially for his patience in explaining the design and answering endless questions about it.

References

[1] ACL2 Web site, `http://www.cs.utexas.edu/users/moore/acl2/`.

[2] Bryant, Randal E. and Yirng-An Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams", *Proceedings of the 32nd Design Automation Conference*, San Francisco, Calif., June 1996.

[3] Clarke, Edmund M., Steven M. German, and Xudong Zhou, "Verifying the SRT Division Algorithm Using Theorem Proving Techniques", *Formal Methods in System Design*, 14:1, January 1999.
`http://www-2.cs.cmu.edu/~modelcheck/ed-papers/VtSRTDAU.pdf`

[4] Gerwig, G., H. Wetter, E.M. Schwarz, J. Haess, C.A. Krygowski, B.M. Fleischer, and M. Kroener, "The IBM eServer z990 floating-point unit", *IBM Journal of Research and Development*, Volume 48, Number 3/4, 2004.
`http://www.research.ibm.com/journal/rd/483/gerwig.html`

[5] Kapur, Deepak and M. Subramaniam, "Mechanizing Verification of Arithmetic Circuits: SRT Division", Invited Talk, Proc. FSTTCS-17, Kharagpur, India, Springer LNCS 1346, pp. 103-122, Dec 1997.
`http://www.cs.unm.edu/~kapur/myabstracts/fsttcs97.html`

[6] Parhami, Behrooz, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.

[7] Pratt, V., "Anatomy of the Pentium Bug", *TAPSOFT '95: Theory and Practice of Software Development*, LNCS 915, Springer-Verlag, May 1995.
`https://eprints.kfupm.edu.sa/25851/1/25851.pdf`

[8] Ruess, Harald and Natarajan Shankar, “Modular Verification of SRT Division”, *Formal Methods in System Design*, 14:1, January 1999.
`http://www.csl.sri.com/papers/srt-long/srt-long.ps.gz`

[9] Robertson, J.E., “A New Class of Digital Division Methods”, *IRE Transactions on Electronic Computers*, Vol. EC-7, 1958.

[10] Russinoff, David M., “A Formal Theory of Register-Transfer Logic and Computer Arithmetic”, `http://www.russinoff.com/libman/`.

[11] Russinoff, David M., “Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover”, IMACS World Congress, Paris, 2005.
`http://www.russinoff.com/papers/paris.html`.

[12] Taylor, G.S., “Compatible Hardware for Division and Square Root”, *Proceeding of the 5th Symposiom on Computer Arithmetic*, IEE Computer Society Press, 1981.

[13] Tocher, K.D., “Techniques of Multiplication and Division for Automatic Binary Computers”, *Quarterly Journal of Mechanics and Applied Mathematics*, Vol. 2, 1958.

Appendix A: XFL Definition of ϕ

```
int phi(nat i, nat j) {
  switch (i) {
  case 0:
    switch (j) {
    case 0x09: case 0x08: case 0x07: case 0x06: case 0x05:
      return 3;
    case 0x04: case 0x03:
      return 2;
    case 0x02: case 0x01:
      return 1;
    case 0x00: case 0x1F: case 0x1E:
      return 0;
    case 0x1D: case 0x1C:
      return -1;
    case 0x1B: case 0x1A:
      return -2;
    case 0x19: case 0x18: case 0x17: case 0x16: case 0x15:
      return -3;
    default: assert(false);
    }
  case 1:
    switch (j) {
    case 0x0B: case 0x0A: case 0x09: case 0x08:  case 0x07:  case 0x06:
      return 3;
    case 0x05: case 0x04: case 0x03:
      return 2;
    case 0x02: case 0x01:
```

```
    return 1;
  case 0x00: case 0x1F: case 0x1E:
    return 0;
  case 0x1D: case 0x1C:
    return -1;
  case 0x1B: case 0x1A: case 0x19:
    return -2;
  case 0x18: case 0x17: case 0x16: case 0x15:  case 0x14:  case 0x13:
    return -3;
  default: assert(false);
  }
case 2:
  switch (j) {
  case 0x0D: case 0x0C: case 0x0B: case 0x0A: case 0x09: case 0x08:
    return 3;
  case 0x07: case 0x06: case 0x05:
    return 2;
  case 0x04: case 0x03: case 0x02: case 0x01:
    return 1;
  case 0x00: case 0x1F: case 0x1E:
    return 0;
  case 0x1D: case 0x1C: case 0x1B:
    return -1;
  case 0x1A: case 0x19: case 0x18: case 0x17:
    return -2;
  case 0x16: case 0x15: case 0x14: case 0x13:  case 0x12:  case 0x11:
    return -3;
  default: assert(false);
  }
case 3:
  switch (j) {
  case 0x0F: case 0x0E: case 0x0D: case 0x0C:
  case 0x0B: case 0x0A: case 0x09: case 0x08:
    return 3;
  case 0x07: case 0x06: case 0x05:
    return 2;
  case 0x04: case 0x03: case 0x02: case 0x01:
    return 1;
  case 0x00: case 0x1F: case 0x1E:
    return 0;
  case 0x1D: case 0x1C: case 0x1B:
    return -1;
  case 0x1A: case 0x19: case 0x18: case 0x17:
    return -2;
  case 0x16: case 0x15: case 0x14: case 0x13:
  case 0x12: case 0x11: case 0x10:
    return -3;
  default: assert(false);
  }
  default: assert(false);
  }
}

// The table that is actually used by the implementation contains
// only non-negative entries; the sign is computed separately:
nat SRTLookup(nat i, nat j) {
  return abs(phi(i, j));
}
```

Appendix B: XFL Model of the Implementation

```
// The function SRT is an XFL model of the Llano integer divider.  It
// has four input parameters:
//   (1) isSigned: a boolean indication of whether the dividend, divisor,
//       quotient, and remainder are represented as signed or unsigned
//       integers.
//   (2) w: the width of the divisor, quotient, and remainder, which
//       may be 8, 16, 32, or 64; the width of the dividend is 2*w.
//   (3) XEnc: the encoding of the divisor.
//   (4) YEnc: the encoding of the dividend.
// Three values are returned:
//   (1) A boolean indication of successful completion, which is false if
//       either the divisor is zero or the quotient is too large to be
//       represented in the indicated format.  The other two values are
//       invalid in this event.
//   (2) The encoding of the quotient.
//   (3) The encoding of the remainder.

<bool, nat, nat> SRT(nat YEnc, nat XEnc, nat w, bool isSigned) {
  assert((w == 8) || (w == 16) || (w == 32) || (w == 64));

  // Division by 0 signals an error:
  if (XEnc == 0) {
    return <false, 0, 0>;
  }

  // The following variables appear in assertions but are not involved
  // in the computation of the function values:
  int Y;  // value of dividend
  int X;   // value of divisor
  int QPart;  // value of quotient during iteration
  int QPre;  // value of quotient before fix-up
  int RPre; // value of remainder before fix-up
  int Q;  // value of quotient after fix-up
  int R; // value of remainder after fix-up
  int m;  // value derived from table, -3 <= m <= 3
  rat d;  // shifted divisor, 1 <= abs(d) < 2
rat p;  // partial remainder, abs(p) <= abs(d)
nat i;  // first argument of phi
nat j;  // second argument of phi

// Decode operands:
if (isSigned) {
  Y = SgndIntVal(2*w, YEnc[2*w-1:0]);
  X = SgndIntVal(w, XEnc[w-1:0]);
}
else {
  Y = YEnc[2*w-1:0];
  X = XEnc[w-1:0];
}

// Compute the number of divisor bits that follow the leading sign
// bits.  In the case of the negative of a power of 2, the trailing
// sign bit is included as a divisor bit:
bool XSign = isSigned ? XEnc[w-1] : false;
nat b = w;
while ((b > 0) && (XEnc[b-1] == XSign)) {
  b--;
}
bool XNegPower2 = XSign && ((b == 0) || (XEnc[b-1:0] == 0));
nat XNB = XNegPower2 ? b+1 : b;
assert(XNB == expo(X) + 1);

// Compute dEnc, a bit vector encoding of d = X >> expo(X):
nat dEnc = 0;
dEnc[67] = XSign;
dEnc[66] = XSign;
dEnc[65:66-XNB] = XEnc[XNB-1:0];
d = X >> expo(X);
assert(dEnc == d[2:-65]);
// Compute leading 2 bits of fractional part of d:
nat dIndex;
if (XSign == 0) {
  dIndex = dEnc[64:63];
}
else if (XNegPower2) {
  dIndex = 0;
}
else if (dEnc[62:0] == 0) {
  dIndex = ((~dEnc[64] | ~dEnc[63]) << 1) | dEnc[63];
}
else {
  dIndex = ~dEnc[64:63];
}
i = dIndex; // first argument of phi
assert(i == fl(4*(abs(d) - 1)));
```

```
// Compute the number of dividend bits that follow the leading sign
// bits.  In the negative case, the trailing sign bit is included as
// a dividend bit.
bool YSign = isSigned ? YEnc[2*w-1] : false;
b = 2*w;
while ((b > 0) && (YEnc[b-1] == YSign)) {
  b--;
}
nat YNB = YSign ? b + 1 : b;
if (Y > 0) {
  assert(1 << (YNB - 1) <= Y && Y < 1 << YNB);
}
else if (Y < 0) {
  assert(1 << (YNB - 2) < abs(Y) && abs(Y) <= 1 << (YNB - 1));
};
assert(Y == 0 || YNB >= expo(Y)+1);

// Compute number of iterations:
nat n;
if (YNB >= XNB) {
  n = fl((YNB - XNB)/2) + 1;
}
else {
  n = 0;
}
assert(abs(Y) <= abs(X) << 2*n);

// Initialize pEncHi, pEncLo, and carryHi, which form a redundant
// representation of the partial remainder:
nat pEnc = 0;
pEnc[131] = YSign;
pEnc[130] = YSign;
pEnc[129] = YSign;
if (YNB != 0) {
  if (YNB[0] == XNB[0]) {
    pEnc[128] = YSign;
    pEnc[127:128-YNB] = YEnc[YNB-1:0];
  }
  else {
    pEnc[128:129-YNB] = YEnc[YNB-1:0];
  }
}
nat pEncHi = pEnc[131:64];
nat pEncLo = pEnc[63:0];
nat carryHi = 0;
assert(n >= 32 || pEncLo[63-2*n:0] == 0);
p = Y >> (expo(X) + 2*n); // initial partial remainder
if (YNB >= XNB) {
  assert(((pEncHi << 64) | pEncLo) == p[2:-129]);
}
else if (YNB[0] == XNB[0]) {
  assert(pEncHi == (Y << (64 - YNB))[67:0]);
}
else {
  assert(pEncHi == (Y << (65 - YNB))[67:0]);
}

// Initialize the quotient:
QPart = 0;  // partial quotient
nat Q0Enc = 0;  // encoding of QPart
nat QPEnc = 1;  // encoding of QPart+1
nat QMEnc = 0x7FFFFFFFFFFFFFFF;  // encoding of QPart-1

// On each iteration, the next partial remainder is computed and the
// quotient is updated:
for (nat k=1; k<=n; k++) {

  // Table lookup:
  nat pTop = pEncHi[67:62];
  bool pSign = pTop[5];
  nat pIndex = pTop[4:0]; // second argument of SRTLookup
  nat mAbs = SRTLookup(dIndex, pIndex);
  bool mSign = XSign ^ pSign;
  m = mSign ? -mAbs : mAbs;
  if (pTop == 0x2F) {
    j = 0x10;
  }
  else {
    j = pIndex;
  }
  assert(m == (XSign ? -phi(i, j) : phi(i, j)));
  assert(SgndIntVal(5, j)/8 <= p && p < SgndIntVal(5, j)/8 + 1/4);

  // 4*p - dm is computed as a sum of four terms.
  // The first two, addA and addB, represent -d*m:
  nat addA;
  if (mAbs[1] == 0) {
    addA = 0;
  }
  else if (mSign) {
    addA = dEnc[66:0] << 1;
  }
  else {
    addA = (~dEnc[66:0] << 1) | 1;
  }
  nat addB;
  if (mAbs[0] == 0) {
```

```
  addB = 0;
}
else if (mSign) {
  addB = dEnc;
}
else {
  addB = ~dEnc[67:0];
}
// A correction term is required to complete the 2's complement
// in case m > 0:
nat inject = 0;
if (!mSign) {
  if (mAbs[0] ^ mAbs[1]) {
    inject = 1;
  }
  else if (mAbs[0] & mAbs[1]) {
    inject = 2;
  }
}
assert((addA + addB + inject)[67:0] ==  (-d*m)[2:-65]);

// addC and addD represent the upper bits of 4*p:
nat addC = (pEncHi << 2) | pEncLo[63:62];
nat addD = (carryHi << 2) | inject;
assert((addC + addD - inject)[67:0] == (4*p)[2:-65]);

// The next partial remainder:
p = 4*p - m*d;
assert(abs(p) <= abs(d));
assert((addA + addB + addC + addD)[67:0] == p[2:-65]);

// 4-2 compression:
nat sum1 = addA ^ addB ^ addC;
nat carry1 = (addA & addB | addB & addC | addA & addC) << 1;
nat sum2 = sum1 ^ carry1 ^ addD;
nat carry2 = (sum1 & carry1 | carry1 & addD | sum1 & addD) << 1;
assert((sum2 + carry2)[67:0] == p[2:-65]);

// Update the redundant representation of p:
pEncHi = ((sum2[67:62] + carry2[67:62])[5:0] << 62) | sum2[61:0];
pEncLo = (pEncLo << 2)[63:0];
carryHi = carry2[61:0];
assert((((pEncHi + carryHi)[67:0] << 64) | pEncLo) == p[2:-129]);
assert(n >= k + 32 || pEncLo[63-2*(n-k):0] == 0);

// Update quotient:
QPart = 4*QPart + m;
assert(abs(QPart) < (1 << 2*k));
if (mAbs == 0) {
    QPEnc = (Q0Enc << 2)[66:0] | 1;
    QMEnc = (QMEnc << 2)[66:0] | 3;
    Q0Enc = (Q0Enc << 2)[66:0];
  }
  else if (mSign == 0) {
    switch (mAbs) {
    case 1:
      QPEnc = (Q0Enc << 2)[66:0] | 2;
      QMEnc = (Q0Enc << 2)[66:0];
      Q0Enc = (Q0Enc << 2)[66:0] | 1;
      break;
    case 2:
      QPEnc = (Q0Enc << 2)[66:0] | 3;
      QMEnc = (Q0Enc << 2)[66:0] | 1;
      Q0Enc = (Q0Enc << 2)[66:0] | 2;
      break;
    case 3:
      QPEnc = (QPEnc << 2)[66:0];
      QMEnc = (Q0Enc << 2)[66:0] | 2;
      Q0Enc = (Q0Enc << 2)[66:0] | 3;
      break;
    default: assert(false);
    }
  }
  else { // mSign == 1
    switch (mAbs) {
    case 1:
      QPEnc = (Q0Enc << 2)[66:0];
      Q0Enc = (QMEnc << 2)[66:0] | 3;
      QMEnc = (QMEnc << 2)[66:0] | 2;
      break;
    case 2:
      QPEnc = (QMEnc << 2)[66:0] | 3;
      Q0Enc = (QMEnc << 2)[66:0] | 2;
      QMEnc = (QMEnc << 2)[66:0] | 1;
      break;
    case 3:
      QPEnc = (QMEnc << 2)[66:0] | 2;
      Q0Enc = (QMEnc << 2)[66:0] | 1;
      QMEnc = (QMEnc << 2)[66:0];
      break;
    default: assert(false);
    }
  }
  assert(Q0Enc == QPart[66:0]);
  assert(QMEnc == (QPart - 1)[66:0]);
  assert(QPEnc == (QPart + 1)[66:0]);
}
```

```
// Remainder and quotient before fix-up:
RPre = p << expo(X);
QPre = QPart;
assert(Y == RPre + QPre*X);
assert(abs(RPre) <= abs(X));

// Encoding of remainder:
nat REncPre = (pEncHi + carryHi)[66:0];
if (YNB >= XNB) {
  assert(REncPre == (RPre << (66 - XNB))[66:0]);
}
else if (YNB[0] == XNB[0]) {
  assert(REncPre == (RPre << (64 - YNB))[66:0]);
}
else {
  assert(REncPre == (RPre << (65 - YNB))[66:0]);
}

// Fix-up is required if either the remainder and the dividend have
// opposite signs or the absolute value of the remainder is the same
// as that of the divisor.  The signals RIs0, RPosX, and RNegX, which
// indicate whether the remander is 0, X, or -X, are computed in parallel
// with the the addition and may not refer to the sum:

bool RSignPre = REncPre[66];
bool RIs0 = (pEncHi[66:0] ^ carryHi ^ ((pEncHi[65:0] | carryHi) << 1)) == 0;
assert(RIs0 == (RPre == 0));

nat RPosXSum = pEncHi[66:0] ^ carryHi ^ ~dEnc[66:0];
nat RPosXCarry = (pEncHi[66:0] & carryHi |
                      pEncHi[66:0] & ~dEnc[66:0] |
                      carryHi & ~dEnc[66:0]) << 1;
bool RPosX = (RPosXSum ^ RPosXCarry ^
              (((RPosXSum[65:0] | RPosXCarry[65:0]) << 1) | 1)) == 0;;
assert(RPosX == (RPre == X));

nat RNegXSum = pEncHi[66:0] ^ carryHi ^ dEnc[66:0];
nat RNegXCarry = (pEncHi[66:0] & carryHi |
                      pEncHi[66:0] & dEnc[66:0] |
                      carryHi & dEnc[66:0]) << 1;
bool RNegX = (RNegXSum ^ RNegXCarry ^
              ((RNegXSum[65:0] | RNegXCarry[65:0]) << 1)) == 0;;
assert(RNegX == (RPre == -X));

bool RIsX = RPosX | RNegX;
assert(RIsX == (abs(RPre) == abs(X)));

bool fixupNeeded = RIsX || (!RIs0) && (RSignPre != YSign);
```

```
nat REnc; // final encoding of remainder
if (!fixupNeeded) {
  REnc = REncPre[65:2];
  R = RPre;
}
else if (RIsX) {
  REnc = 0;
  R = 0;
}
else if (RSignPre == XSign) {
  REnc = ((REncPre[65:2] << 2) + ~dEnc[65:0] + 1)[65:2];
  R = RPre - X;
}
else {
  REnc = ((REncPre[65:2] << 2) + dEnc[65:0])[65:2];
  R = RPre + X;
}
bool RSign = YSign & ~RIs0[0] & ~RIsX[0];
if (YNB >= XNB) {
  assert(REnc == (R << (64 - XNB))[63:0]);
}
else if (YNB[0] == XNB[0]) {
  assert(REnc == (R << (62 - YNB))[63:0]);
}
else {
  assert(REnc == (R << (63 - YNB))[63:0]);
}

nat QEnc; // final encoding of quotient
if (!fixupNeeded) {
  QEnc = Q0Enc;
  Q = QPre;
}
else if (RSignPre == XSign) {
  QEnc = QPEnc;
  Q = QPre + 1;
}
else {
  QEnc = QMEnc;
  Q = QPre - 1;
}
assert(Y == R + Q*X);
assert(abs(R) < abs(X));
assert((R == 0) || ((R < 0) == (Y < 0)));
assert(n > 33 || QEnc == Q[66:0]);

// Determine whether the quotient is representable:
bool QTooLarge;
if (YNB > XNB + w) {
    QTooLarge = true;
  }
  else if (n <= 1) {
    QTooLarge = false;
  }
  else if (YSign == XSign) {
    QTooLarge = (QEnc[w+2:w] != 0) || isSigned && (QEnc[w-1] != 0);
  }
  else {
    QTooLarge = (QEnc[w+1:w-1] != 7);
  }
  if (isSigned) {
    assert(QTooLarge == ((Q > MaxSgndIntVal(w)) ||
                         (Q < MinSgndIntVal(w))));
  }
  else {
    assert(QTooLarge == (Q >= 1 << w));
  }

  if (QTooLarge) {
    return <false, 0, 0>;
  }

  // Compute the final results:
  nat QOut = QEnc[w-1:0];
  nat ROut;
  if (YNB >= XNB) {
    ROut = ((RSign << w) - (RSign << XNB)) | REnc[63:64-XNB];
  }
  else if (YNB[0] == XNB[0]) {
    ROut = ((RSign << w) - (RSign << (YNB+2))) | REnc[63:62-YNB];
  }
  else {
    ROut = ((RSign << w) - (RSign << (YNB+1))) | REnc[63:63-YNB];
  }
  assert(QOut == Q[w-1:0]);
  assert(ROut == R[w-1:0]);
  return <true, QOut, ROut>;
}
```