Verification of the Decimal Floating-Point Square Root Operation

Amr Sayed Ahmed *, Hossam Fahmy† and Ulrich Kuehne*
*Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
Email: {asahmed, ulrichk}@informatik.uni-bremen.de
† Electronics and Communications Department, Cairo University, Giza, Egypt
Email: hfahmy@stanfordalumni.org.com

***Abstract*— Decimal floating-point is a relatively recent addition to the IEEE standard (IEEE Std 754-2008). There exist few verification techniques that can check whether software libraries or hardware designs are in compliance with the standard. Our work presents a verification process to verify implementation of the decimal floating-point square root operation. We present an effective simulation based verification technique using test cases that verify the corner cases of the operation. The test cases are generated by solving constraints describing these corner cases with a dedicated constraint solver. We use the engine also to find the extreme hardest-to-round cases. The engine proved its usefulness by finding severe bugs in two well-tested designs.**

I. Introduction

Decimal floating-point implementations – be it in software libraries or hardware designs – have many advantages over binary floating-point especially in financial and commercial applications [1]. Decimal floating-point is defined in the latest IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008) [2]. To verify compliance with this standard, new verification techniques are needed.

The method that we use is simulation based verification, a simulation method based on coverage models that checks the corner cases of a decimal floating-point operation. The method guarantees that the tests cover the hard corner cases of the operation. In contrast to our dedicated tests, pure random simulation does not guarantee a good coverage for these corner cases due to the large problem space. Also to the best of our knowledge, formal verification techniques have not yet been applied on any decimal floating-point implementations. For some operations, verification methods have been developed based on coverage models and constraint solving [3], [4], [5]. Here, we show how the method can be adapted to verify implementations of the decimal square root (DSQRT) floating point operation, which has not been considered so far.

The coverage model that our method is based on, consists of a set of tasks, where each task represents a certain case of the operation in form of constraints. The constraints can be defined wrt. the input, the result, or the intermediate result of the DSQRT operation. Here, the intermediate result is the exact result of the operation without any bound, i.e. the result before rounding. We represent the operation by nonlinear equations. The elements of these equations are the decimal digits of the operation input or the operation intermediate result, representing the relation between the input digits and the intermediate result digits. Each targeted case is represented as range constraints on the digits of the equations of the operation. These range constraints are solved by a software engine. If a solution is found, it generates a test vector to verify the case in a DSQRT floating point design using simulation.

To the best of our knowledge, our work is the first to verify the DSQRT floating-point operation. We have designed an engine with new algorithms to solve DSQRT simultaneous constraints on input and on the unbounded intermediate result. We have also implemented coverage models targeting different corner cases according to our understanding of the standard and the DSQRT operation. The strength in our contribution is the ability of the engine to solve simultaneous constraints on input and on the unbounded intermediate result in practical time, especially the ability of solving constraints on the patterns of zeros and nines in the intermediate result significand, that we use to find the hardest-to-round cases of the operation. We found that Only $2p-1$ digits not including leading zeros are sufficient to do the correct rounding for a Decimal Floating-Point Square Root operation with $p > 6$.

II. Overview of the square root engine

In our work, decimal floating-point number is defined as $(-1)^s(d_{p-1}d_{p-2}d_{p-3}\cdots d_0)10^q$, where s is the *sign*, the term $d_{p-1}d_{p-2}d_{p-3}\cdots d_0$ is called the *significand* of the number, where each $d_i \in [0,9]$, and q is called the *exponent*. The FP standard [2] defines the *precision* p as the maximum number of digits in the significand.

The inverse operation of the square root is the multiplication of the intermediate result with itself, which gives the input of the square root operation. The engine is based on solving the non linear equations that result from multiplying the intermediate result with itself. We can extract these non linear equations from Figure 1, where each column represents one nonlinear equation. The figure shows an example of the squarer of the intermediate result for $p = 8$ where z_j denotes the intermediate result digit of weight 10^j, and x_i denotes the input digit of weight 10^i.

The main process in the engine is finding the intermediate result's significand z and the input significand x that satisfy the constraints. In this process, the engine solves the range constraints on each digit x_i and z_j by using one of two algorithms. The first one is called (MSC) and is used to solve the range constraints on the most significant p digits of the intermediate result significand and the all digits of the input significand, which includes the cases with exact result. The second one is called (LSC) and solves the constraints on the least significant digits that follow the highest p digits of the intermediate result's significand, especially, the series of zeros or nines in the least significant digits of z that are followed by a non zero digit, which are frequently needed to verify the rounding cases.

As shown in Figure 1, the general form of the nonlinear equations from the column of index i is one of the following two forms.

$$br_i = x_i + 10 * br_{i+1} - \sum_{j=i-w_x/2}^{w_x/2} z_{i-j} * z_j \qquad (1)$$

$$cr_i = cr_{i-1}/10 + \sum_{j=i-w_x/2}^{w_x/2} z_{i-j} * z_j - x_i, \qquad (2)$$

where $w_x \leq i \leq (-2p-3+w_x)$, br_i is the borrow that propagates between the columns from left to right (highest index to lowest index), and cr_i is the carry that propagates between the columns from right to left (lowest index to highest index). By substituting with the values of i in Equation 1 or 2, we get $(2p-1)$ nonlinear equations that are needed to build the relations between the $(2p-1)$ digits of the intermediate result and the digits of the input. Each digit in these equations has its constraint range, which for each digit x_i with $w_x \leq i \leq 1$ is defined by $0 \leq x_i \leq 9$, or is defined by the task .

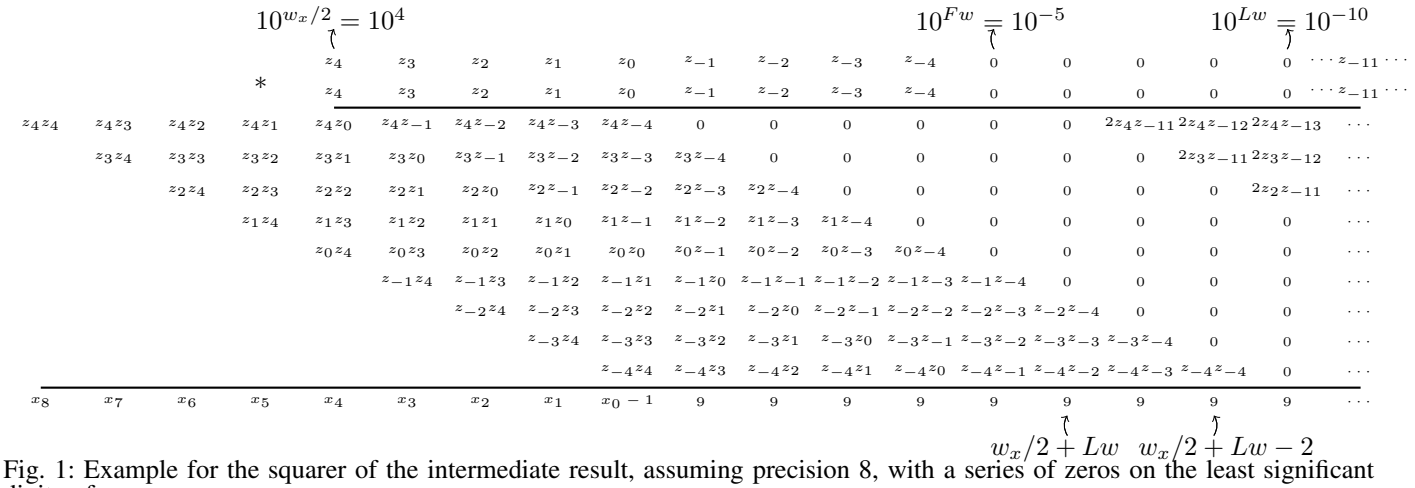

Fig. 1: Example for the squarer of the intermediate result, assuming precision 8, with a series of zeros on the least significant digits of z

In the same way, the constraint range for each digit z_j with $w_x/2 \leq j \leq w_x/2 - 2p + 2$.

The engine transforms the problem to a satisfiability problem. It tries to find an assignment for the operation digits (x_i, z_j) that satisfies the nonlinear equations and the constraints interval. It chooses the MSC algorithm or the LSC algorithm according to the type of constraints.

The MSC and LSC algorithms separate the nonlinear equations and the constraints intervals into groups and try to find satisfiable assignments for the unassigned digits of each group, by adding equations to the group that represent the dependency of the unassigned digits on other nonlinear equations outside the currently group, and performing a local search. The useful of theses added equations is pruning the space of the local search. If no solution could be found, a backtrack is performed to the previous group. The two algorithms differ in the ordering of the groups and in the added equations to each group.

The MSC algorithm solves the nonlinear equations of indices $w_x \leq i \leq (-2p - 3 + w_x)$ in the form of equation 1 and orders them from the highest index of i.Then it puts three equations in a group, such that the first group has the first, second, and the third equations, while the second group has the second, the third, and the fourth equations, and so on until all the equations are assigned in groups.

In The MSC algorithm, the added equations to every group, are the equations that represent the dependency of the digit br_{i-2} on the next three equations that follow the group being solved. So that the algorithm substitutes with the minimum and the maximum values of the unassigned digits in these equations to get six new equations. These new equations represent the minimum and the maximum value of br_{i-2}, are added to the group during performing the local search on the unassigned digits of this group.

While for the LSC algorithm, as shown in Figure 1, the intermediate result's significand z has a series of zeros. Due to this series of zeros, the elements are decreased in the columns of the lowest indices. The LSC algorithm exploits this reduction in these columns elements, and solves the nonlinear equations of these columns in the form of equation 2, from right (lowest index) to left. It orders these equations from the lowest index and divides them into groups in the same way as in the MSC algorithm. It adds to each group before performing the local search, the carry equations $(cr_i) mod_{10} = 0$, for each carry in the equations group.

III. Experimental results

The engine generates the test vectors in two formats of the IEEE standard: Decimal64 and Decimal128. The time to generate one test vector depends on the constraints that have been solved and the factor of randomization that the engine needs. In our experiments, the engine solves these constraints in practical time as shown in Table I. The table shows the maximum and the minimum times that the engine needed to solve a task of the existing constraints and generate one test vector.

TABLE I: The Time Performance of The Engine

Test vector Format	Minimum Time	Maximum Time
Decimal 64	0.006 seconds	37 seconds
Decimal 128	0.017 seconds	2.35 minutes

The experiments have been conducted on an Intel(R) Pentium(R) 4 CPU 3.20GHZ with g++ (Ubuntu 4.4.3) compiler. The number of generated test vectors can be adapted as needed. In the experiment, the engine solved the coverage models one time and generated about 57,000 test vectors for Decimal64 and about 199,000 test vectors for Decimal128. The test vectors are available in [6].

Although the engine solves constraints on the input and the intermediate result only, it managed to discover subtle bugs in in two designs by solving constraints on patterns of zeros and nines in the intermediate result significand for hard rounding cases. The two designs are the DecNumber library [8] and a Silminds hardware design [7].

IV. Conclusion

This paper presents the first work for the verification of the decimal square root operation. It is based on solving non linear constraints using a dedicated software engine and generating test vectors to verify implementations of the square root decimal floating-point operation. The test vectors have proven their efficacy in discovering subtle bugs in an open source library and a commercial floating point implementation.

References

[1] A. Fahmy, R. Raafat, A. Abdel-Majeed, R. Samy, T. ElDeeb, and Y. Farouk, "Energy and delay improvement via decimal floating point units," in *IEEE Symposium on Computer Arithmetic*, 2009, pp. 221–224.

[2] *IEEE 754-2008, Standard for Floating-Point Arithmetic*, 2008.

[3] E. Guralnik, M. Aharoni, A. J. Birnbaum, and A. Koyfman, "Simulation-based verification of floating-point division," *IEEE Trans. Computers*, vol. 60, no. 2, pp. 176–188, 2011.

[4] A. Sayed-Ahmed, H. Fahmy, and M. Hassan, "Three engines to solve verification constraints of decimal floating-point operation," in *Signals, Systems and Computers (ASILOMAR)*, 2010, pp. 1153–1157.

[5] A. Sayed-Ahmed, A. Fahmy, and R. Samy, "Verification of decimal floating-point fused-multiply-add operation," in *Computer Systems and Applications (AICCSA)*, 2011, pp. 255–262.

[6] The Arithmetic operations debugging and verification, http://eece.cu.edu.eg/ hfahmy/arith_debug/, 2014.

[7] R. Raafat, A. Mohamed, H. Fahmy, Y. Farouk, M. Elkhouly, T. Eldeeb, and R. Samy. "Decimal floating-point square-root unit using Newton-Raphson iterations," July 2011. US Patent application number 13177488.

[8] "The DecNumber library version 3.68," 2013, available at http://speleotrove.com/decimal/decnumerr.html.