SPEAR: Hardware-based Implicit Rewriting for Square-root Circuit Verification

Atif Yasin*, Tiankai Su*, Sébastien Pillement†, Maciej Ciesielski*
*University of Massachusetts, Amherst, MA, USA
†Univ Nantes, CNRS, IETR UMR 6164, F-44000 Nantes, France
ayasin@umass.edu, tiankaisu@umass.edu, sebastien.pillement@univ-nantes.fr, ciesiel@umass.edu

***Abstract*—The paper addresses the formal verification of gate-level square-root circuits. Division and square root functions are some of the most complex arithmetic operations to implement and proving the correctness of their hardware implementation is of great importance. In contrast to standard approaches that use satisfiability and equivalence checking techniques, the presented method verifies whether the gate-level square-root circuit actually performs a root operation, instead of checking equivalence with a reference design. The method extends the algebraic rewriting technique developed earlier for multipliers and introduces a novel technique of implicit *hardware rewriting*. The tool called SPEAR based on hardware rewriting enables the verification of a 256-bit gate-level square-root circuit with 0.26 million gates in under 18 minutes.**

I. Introduction

Considerable progress has been made in recent years in verification of arithmetic circuits, such as multipliers, fused multiply-add, multiply-accumulate, and other components of arithmetic datapaths [1][2]. However, formal verification of square-root circuits has received only a limited attention. A notable exception are theorem provers, inductive, non-automated systems, which concentrate on proving the correctness of the arithmetic algorithms of the designs and the resulting architectures rather than of the low-level hardware implementations [3].

Square-root computation plays a major role in many domains, including computer arithmetic, computational geometry, embedded systems, and other special purpose applications. It belongs to the class of dividers and is one of the most complex arithmetic operation to implement that requires careful hardware implementation and verification [3]. Square-root (SQRT) computation has numerous applications, including Euclidean Norm as well in the generalizations of Hilbert Spaces. It defines an important concept of standard-deviation (root of a variance), and has a major application in quadratic formula to compute roots for quadratic equations and fields. This work concentrates on combinational square-root circuits, both integer and floating point (implemented in fractional arithmetic). The proposed verification approach extends the algebraic rewriting model, originally proposed and successfully applied to integer and Galois Field multipliers [1] [4], to hardware rewriting applicable to SQRT verification.

The rest of the paper is organized as follows. Section II provides the necessary background and reviews the related work in the field. Section III reviews the algebraic rewriting technique used in previous work on verification that motivated this work. Section IV describes the mathematical foundation of the integer and floating point (fractional) SQRT verification, while Section V develops the hardware-based rewriting technique. Finally, Section VI presents the preliminary results and conclusions.

II. Background and Related Work

A popular technique employed in industry in arithmetic circuit verification is Theorem Proving. Theorem provers are inductive reasoning system that use mathematical formulation to verify the correctness of an arithmetic algorithm and its architecture. These systems rely on a set of rewriting rules and complex formulas to represent the circuit and require a domain-specific user knowledge [5][6]. The success of the proof relies on the choice of the rules and on the order in which they are applied to the system, with no guarantee of a successful conclusion. Theorem proving has been also used in square-root verification [3]. Other verification techniques use SAT-based approach, equivalence checking and various canonical diagrams, such as BDDs [7] and BMDs [8]. Some have been useful in proving floating point multipliers [9], but the literature is rather scarce on square-root circuit verification.

An approach to formal verification of arithmetic circuits that emerged in recent years is based on computer algebra. In this approach, the specification of an arithmetic function and its implementation are represented as polynomial rings in a given field. The verification problem is then posed as checking if the implementation satisfies the specification using canonical Groëbner basis, known in the computer algebra community as the *ideal membership testing* [10]. Several modifications have been reported in the literature that improve the efficiency of the technique, both for integer and Galois Field multipliers [11][2]. However, those techniques have not been successful in verifying gate-level dividers and square-root circuits due to the memory explosion problem.

An alternative approach to an arithmetic verification of gate-level circuits has been proposed in [1][12], using algebraic rewriting of the specification polynomial. With this approach, the polynomial representing encoding of the primary outputs (called the *output signature*) is transformed by a series of rewriting steps into a polynomial expressed in terms of the primary inputs (the *input signature*). The transformation uses

978-3-9819263-4-7/DATE20/©2020 EDAA

algebraic models of circuit elements, such as logic gates or bit-level arithmetic modules. This method has been successfully used to verify complex adders and multipliers [1][11][13][14] It has not been applied, however, to square-root circuits, because of the difficulty of modeling the square-root's specification that would avoid the memory explosion. Recently, some work has been done in the domain of array dividers [15] that could potentially be applied to square root circuits. This method extracts a high-level arithmetic model from the low-level circuit implementation and compares it with an abstract model of the divider using structural matching. However this approach does not explicitly verify an actual arithmetic function. In contrast, the original method described in this paper actually verifies if the circuit performs the square root operation, regardless of its internal structure or architecture.

III. Algebraic Rewriting

The verification approach described here is an extension of the algebraic rewriting method of [1]. The circuit is modeled in an algebraic domain where each gate is represented as a unique polynomial $f[X]$ with binary variables $X = \{x_1, ..., x_n\}$ and integer coefficients. Table I presents algebraic expressions that model the basic Boolean operators. We refer to them as *characteristic expression* or *characteristic functions*.

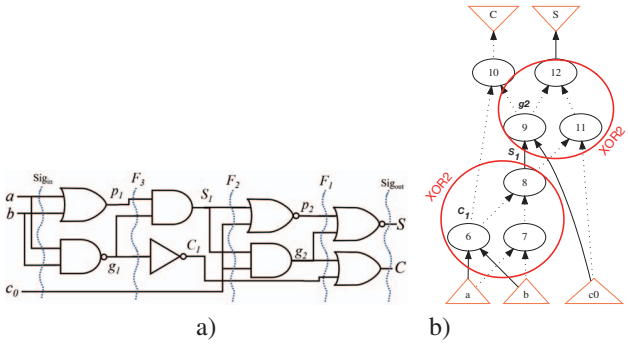
TABLE I: Algebraic models of basic logic operations.

Operation	Boolean model	Algebraic model
$Inv(a)$	$\neg a$	$1 - a$
AND(a,b)	$a \wedge b$	ab
OR(a,b)	$a \vee b$	$a + b - ab$
XOR(a,b)	$a \oplus b$	$a + b - 2ab$

Using these basic models, one can also derive models for complex gates, such as AOI (And-Or-Invert). For example, the algebraic model for logic gate $g = a \vee (b \wedge c)$ can be derived as $g = a + bc - abc$.

Algebraic rewriting starts with an *output signature*, Sig_{out}, a polynomial that represents the binary encoding of the primary outputs. For example, an output signature of an unsigned arithmetic circuit with n output bits is $Sig_{out} = \sum_{i=0}^{n-1} 2^i z_i$. This polynomial is transformed by a series of rewriting steps into an *input signature*, Sig_{in}, the polynomial over the primary input variables, using algebraic models of the circuit components (logic gates). The resulting polynomial is then compared with the specification of the circuit. For example, for a correctly implemented n-bit unsigned multiplier, the input signature is $Sig_{in} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} a_i b_j$, which matches its functional specification, $F = A \cdot B$, with $A = \sum_{i=0}^{n-1} 2^i a_i$, and $B = \sum_{i=0}^{n-1} 2^i b_i$.

Figure 1(a) illustrates the rewriting process of a gate-level arithmetic circuit (full adder, FA) with inputs a, b, c_0. The output signature of the circuit is $Sig_{out} = 2C + S$, determined by the binary encoding of the 2-bit output. The goal is to determine the arithmetic function implemented by this circuit (or, equivalently, to verify if this is a full adder) by rewriting Sig_{out} into an input signature (specification), $Sig_{in} = a + b + c_0$.



a) b)

Fig. 1: Arithmetic circuit (FA): a) gate-level diagram; b) AIG representation

The rewriting is performed by expressing the signals in terms of the characteristic functions of the respective logic gates in reverse-topological order, *e.g.*: $\{S, C, p_2, g_2, S_1, C_1, p_1, g_1\}$. During the rewriting process, some expressions cancel out each other, finally producing the signature at the primary inputs, $Sig_{in} = a + b + c_0$, indicating (or confirming) that this is a full adder.

The rewriting process can be drastically improved by using a functional AIG (And-Invert Graph) representation [16]. This approach identifies major components of half adders, such as XOR and AND/MAJ3, as shown in Figure 1(b). The rewriting over those components drastically reduces the number of rewriting steps and the complexity of the intermediate polynomials.

Several other improvements have been developed in algebraic rewriting, including efficient early reduction of intermediate polynomials [11]. However, these techniques are not directly applicable to the square-root or the divider class of architectures, primarily because the characteristic function of the square-root and the divider circuit is non-linear, making the size of the intermediate polynomials unmanageable.

IV. Square Root Verification

Square root extraction (SQRT) is typically based on a shift and subtract algorithm, and as such the SQRT can be viewed as division with a changing divisor. There are two algorithms that compute the square root using this approach: 1) based on the restoring algorithm, and 2) on the non-restoring algorithm. The two algorithms differ in whether, during each subtraction step, the intermediate value (remainder) is allowed to be negative or not. In a non-restoring algorithm, the final remainder may be negative and can be trivially corrected by subtracting one ulp (unit in least position) from the final result (root). As will be shown later, our technique applies to both types of SQRT circuits, and for both integer and fractional operands.

A. Characteristic Function of SQRT

In order to apply algebraic rewriting to the SQRT circuit, we need to define the input and output signatures for the

circuit and the characteristic function of the square rooter. The obvious inputs and outputs of such a circuit are X (the radicand) and Q (the root). The characteristic function of the SQRT operation, $Q = \sqrt{X}$, can then be described by:

$$X = Q^2 + R, \quad \text{with } R \leq 2Q \tag{1}$$

where R is the remainder (or residue).[1] The remainder R is needed in the expression so that the arithmetic function of the circuit can be represented as a strict equality (characteristic function) rather than an approximation. Its role in Equation (1) is similar to that in the division, $X = QD+R$, where Q is the quotient, D the divisor, and R the remainder. Depending on the square-root extraction algorithm implemented by the circuit, the remainder R can be positive (in the restoring algorithm) or of any sign (in non-restoring algorithm). Using a simple integer example with $X = 13$, the solution to $Q = \sqrt{(13)} \approx 3.6055$ can be either $Q = 3$, $R = 4$; or $Q = 4$, $R = -3$. Typically, $R > 0$ is accepted as a standard solution, but negative remainders can be used, depending on the required precision (this is explained in more detail in Section IV-C.) However, it is important to note that in both cases, equation (1) is satisfied: $X = 13 = 3^2 + 4 = 4^2 - 3 = Q^2 + R$.

Unfortunately, most hardware SQRT implementations do not provide remainder, so in order to use it in our approach, it needs to be generated. Section V-A describes how such a remainder is generated and used in the verification.

B. Integer vs Fractional SQRT

Consider an *integer* SQRT circuit, with radicand X and root Q being integers. To verify the SQRT circuit, we need to prove that the equation $X = Q^2 + R$ is satisfied for every input assignment and that $R \leq 2Q$. It can be shown that R is also integer; and for the restoring algorithm, R is positive and the result Q is unique. In principle, verification of $X = Q^2 + R$ can be achieved by performing algebraic rewriting discussed in Section III, by rewriting $Sig_{out} = Q^2 + R$ at the outputs Q, R, into $Sig_{in} = X$ at the input X. The word-level symbols X, Q, R are represented as polynomials in their bits as variables and with integer coefficients 2^i. Specifically: $X = \sum_{i=0}^{n} 2^i x_i$, $Q = \sum_{i=0}^{n} 2^i q_i$, and $R = \sum_{i=0}^{n} 2^i r_i$. We will return to the verification of the requirement $R \leq 2Q$ later in Section V-B.

It has been shown in [18] and [17] that the integer SQRT circuit will also perform the SQRT function with the *fractional* radicand X and root Q. In fact both circuits use exactly the same algorithm and the same architecture. Such fractional circuits are routinely used for floating point calculations and hence it is important to develop a method for their verification. The two designs will only differ in the representation of the input and the results, with the fractional circuit having binary representation $X = [x_0.x_{-1}....x_{-n}] = \sum_{i=0}^{-n} 2^i x_i$, and similarly for Q. In the fractional case, the number of bits of R is $2n$+1, double that of X, Q. The radicand of the integer circuit is $X = [x_0 x_{-1}....x_{-n}] = \sum_{i=-n}^{0} 2^{i+n} x_i$, where x_0 is the most significant bit and x_{-n} the least significant bit; and similarly for Q and R.

To avoid fractional coefficients, the fractional representation of X and Q can be simply multiplied by 2^n, where n is the even number of bits ($n = 2k$) of X and Q. That is,

$$\sqrt{(}x_0 \ . \ x_{-1}....x_{-n}) = \sqrt{(}x_0 x_{-1}....x_{-n} \cdot 2^{-n}) = 2^{-k}\sqrt{(}x_0 x_{-1}....x_{-n})\cdot$$

This simple normalization gives us the right to apply the algebraic rewriting concept (as well as the hardware rewriting developed later) to fractional SQRT circuits using polynomials with positive coefficients.

C. Restoring vs Nonrestoring SQRT Verification

We close this Section by making remarks about the application of our approach to all versions of SQRT designs, including integer and fractional, both restoring and nonrestoring.

We illustrate this point with the following example for $X = 118/64 = 1.110110$, where $\sqrt{(}118/64) \approx 1.3578....$

Fractional Restoring:
$Q = 86/64 = 1.010110; R = 156/64^2 = 0.000010011100.$
Equation (1) is satisfied: $X = (86/64)^2 + 156/64^2 = 118/64.$

Fractional Nonrestoring: Without correction, $Q = 87/64 = 1.010111$, $R = -17/64^2 = 1.111111101111$ and Equation (1) is satisfied: $X = (87/64)^2 - 17/64^2 = 118/64$. With correction: $Q = 86/64 = 1.010110$, $R = 156/64^2 = 0.000010011100$. In this case Equation (1) is also satisfied: $X = (86/64)^2 + 156/64^2 = 118/64$. This result is obtained by rounding down (truncating) the initial result, while the original one without correction (with $R = -17/64^2$) is actually closer to the real solution and may be more desirable. In any case, regardless of the correction/rounding, the characteristic equation $X = Q^2 + R$ is satisfied and will be used as an invariant in our verification method. Recall that the **integer** solution for $\sqrt{(}118) = \sqrt{(}01110110)_2$, with $Q = 10_{10} = (1010)_2$ and $R = 18 = (10010)_2$, also satisfies Equation (1).

V. Hardware Rewriting for SQRT Verification

A. Remainder Generation

We now come to the critical issue: in order to prove the circuit using equation $X = Q^2 + R$, we need the remainder R. However, a typical SQRT circuit provides only a single output Q. To solve this problem, we restore the "missing" residual output by constructing a circuit $R_{ref} = X - Q_{ref}^2$, with inputs X and Q_{ref}, and output R_{ref}, as shown in Figure 2. Input Q_{ref} is provided by the reference SQRT design, labeled $Ref\sqrt{X}$ in the figure, a golden model known to be correct.

The reconstructed circuit is shown in Figure 3. At this point the rewriting can be done on the modified circuit with the original output Q and the newly generated R_{ref}. However, such a simple-minded, standard rewriting is not efficient, mostly because the output signature $Sig_{out} = Q^2 + R_{ref}$ is nonlinear and the intermediate polynomials can become

[1] The reason for the requirement $R \leq 2Q$ is that for $R \geq 2Q+1$, we have $X = Q^2 + R \geq (Q+1)^2$, hence the result is incorrect [17].

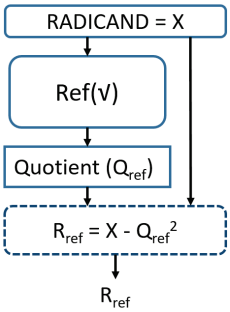


Fig. 2: Residue generation using a Reference Design.

prohibitively large. Our experiments show that the designs with a radicand greater than 8 bits cannot be verified with this approach due to a memory overload. The main culprit seems to be a non-linear signature (Q^2+R), which causes the intermediate polynomial to grow fast, and the rewriting does not converge even with 22 GB of memory.

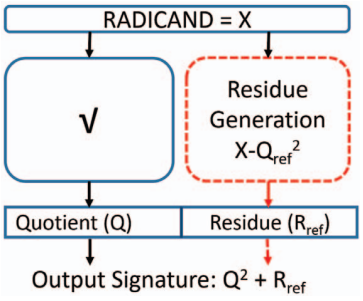


Fig. 3: Conceptual standard rewriting.

The failure of algebraic rewriting is actually not surprising; algebraic rewriting has been successfully applied to complex adders and multipliers, which are characterized by a linear output signature, determined by the encoding of the output bits. For example, a multiplier $X = A \cdot B$ has a clearly defined linear signature $Sig_{out} = X = \sum_{i=0}^{n-1} 2^i x_i$. However, the square-root circuit has a non-linear signature, $X = Q^2+R$. Rewriting a non-linear signature is much more memory intensive, as demonstrated above, and a standard algebraic rewriting proves largely ineffective for these circuits.

The reader may ask at this point, and rightly so: "why not compare Q_{ref} directly to output Q of the design under verification"? We did try it using SAT, but the results were very disappointing, as shown in Table II in Section VI, to be discussed later. The next section describes an original method to solve this problem by extending rewriting to synthesis to be done directly on hardware.

B. Hardware Rewriting

We now introduce the concept of a *Signature Linearizer*, a circuit that transforms a nonlinear signature $Q^2 + R_{ref}$ into a linear one, in an attempt to enable rewriting.

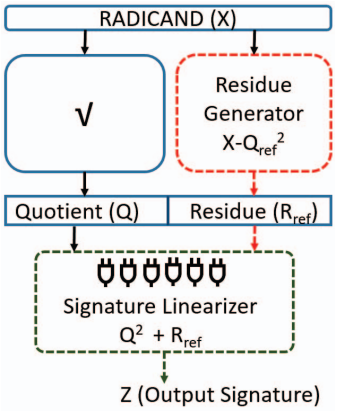


Fig. 4: Hardware rewriting

Figure 4 shows the basic concept of our approach. The upper part of the figure is the circuit that computes $Q^2 + R_{ref}$ described earlier and shown in Figure 3. Since algebraic rewriting of such a polynomial is inefficient, we introduce another circuit that computes function $Z = Q^2+R_{ref}$, derived from the output Q of the original circuit, and the reference remainder R_{ref}, generated from the reference Q_{ref}. Such constructed output Z of the circuit is a *linear* polynomial in its bit variables, $Z = \sum_{i=0}^{n} 2^i z_i$.

The combined circuit has input X and output Z, which for the correct SQRT circuit should satisfy $Z = X$. In principle, this equivalence could be checked by algebraic rewriting of the linear polynomial Z all the way to the primary inputs, to obtain X. However, it turns out that such an algebraic rewriting is still inefficient, since the internal polynomials can become prohibitively large.

Instead, we perform an implicit *hardware rewriting* by resynthesizing the entire circuit that computes Z as a function of X. This resynthesis process uses a state-of-the art synthesis tool, ABC [16] that includes structural and functional hashing ($strash$), functional simplifications ($fraig$), and a final resynthesis step (dch), all using an AIG data structure. In a functionally correct SQRT circuit, the newly constructed circuit should become *redundant* and reduced to a set of wire/buffer connections between X and Z, provided that the added parts (residue generator $R = X - Q^2$ and the linearizer, $Z = Q^2 + R$) are correct.

We cannot, however, rely on resynthesis as a formal proof; if the synthesis does not simplify the design to a redundant state (wires/buffers), we cannot conclude that the circuit is incorrect. In this case, those portions of the circuit that are not reduced to wires can be verified using SAT. Specifically, for each bit Z_i that does not trivially reduce to X_i, we create an XOR/miter and check if the result is unSAT (or, equivalently if the output of an XOR for each pair of bits X_i, Z_i is 0). If the result is unSAT, the circuit is correct; otherwise, the SAT solution provides a counter example that can be used to identify the bug. This hardware "SAT rewriting" idea is illustrated in Figure 5.

It should be emphasized that this verification method is sound only if all parts of the circuit, including the added residual and linearizer circuits, are functionally correct. If the result is unSAT, one may safely conclude that the added circuits are correct as well. The chance of the add-ons being faulty in such a way that they mask the error in the original SQRT circuit is highly unlikely. On the other hand, the presence of an error in any part of the circuit will result in a satisfiable solution to the SAT problem, and it wouldn't be clear if the error comes from the tested circuit or from the added components. However, correctness of these "add-ons" is guaranteed by construction: first the reference remainder, R_{ref}, is constructed from a reference result Q_{ref} of a golden reference SQRT circuit. Then the linearizer circuit $Z = Q^2 + R_{ref}$ is obtained using some golden (certified) squarer Q^2 or a multiplier $Q \cdot Q$ of input Q, and a golden adder that adds Q^2 to R_{ref}. In this case, a satisfiable solution will correctly indicate that the SQRT circuit is faulty, avoiding false negatives.

Similar argument applies to verifying the constraint $R \leq 2Q$ in equation (1), since the circuit for R_{ref} in Figure 2 is derived from a correct reference design Q_{ref}. If this constraint is not satisfied by the SQRT under verification, the output Q of the circuit in conjunction with the correct reference remainder R_{ref} would not match X.

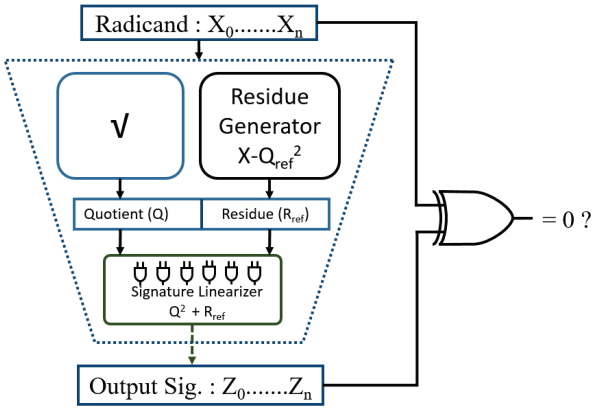


Fig. 5: Final verification using SAT: check if $X_i = Z_i$ ∀i.

The results demonstrate that our methodology works well on both bug-free and buggy designs. Finding a source of a bug and performing the debugging is a separate and challenging problem, which will be considered in future work.

VI. Results

The verification technique described in this paper was implemented in Python and C++ as a stand-alone program SPEAR, which uses ABC [19] at the back-end for synthesis. The program was tested on a number of SQRT circuits with radicand bit-widths varying from 6 to 256. The square-root circuits used in the experiments were generated from Synopsys DWare library and the add-ons (Residual circuit and Linearizer) were generated using the ABC tool [19]. Each design was appended with a residual circuit and a linearizer, and synthesized using ABC. The architecture of the SQRT circuits and the add-ons generated by ABC are different and do not exhibit structural similarities, proving the efficiency of our technique.

In the experiments, circuits with fewer than 24-bit wide radicand, were synthesized down to bare wires, proving that the circuit indeed performs a SQRT function. The larger circuit required formal verification using SAT.

Table II compares the verification time of our technique with those obtained with: standard rewriting; SAT (for equivalence checking between Synopsys DesignWare circuits and the reference design generated by ABC); and simulation. The SAT experiments (column 4) were performed by creating a miter between the SQRT circuit and a reference design obtained from Synopsys DesignWare library and ABC tool respectively. The test for satisfiability was performed using miniSAT [20]. Despite its renown efficiency, miniSAT could not handle circuits with radicand bit-widths greater than 32 bits. Similarly, standard algebraic rewriting of [12] (col 3) could not verify designs beyond 8-bit radicands because of high memory consumption; it used over 22 GB of memory in a matter of minutes.

The table also shows CPU time required for individual parts of our technique, including: the residue generation (col. 6), resynthesis (col. 7), and hardware-rewrite SAT to prove hardware rewriting (col. 8). As we can see from the table, SPEAR outperforms all of the tested techniques.

We also performed extensive simulation experiments. The simulation was unable to handle circuits with more than 24-bit radicand and was aborted after running for over 10 hours of CPU time and consuming over 10 GB of memory for storing the intermediate simulation results.

In contrast to all these schemes, SPEAR was able to verify square-root designs with up to 256-bit radicands, containing over 260,000 gates in less than 18 minutes, while using less than 4 GB of memory.

VII. Conclusions and Future work

This paper presents a novel methodology to verify the gate-level implementation of a square-root circuit using an original *hardware rewriting* technique. Specifically, it verifies the square-root circuit against its *functional specification* and does not require a reference circuit per se, except for generating the reference R_{ref} circuit. That is, we do not explicitly compare the circuit against some reference design, because (as demonstrated by the results) such an equivalence checking problem cannot be solved by the state-of-the art SAT solvers.

It may seem surprising that adding more hardware (R_{ref} and Q^2+R_{ref}) to the design actually simplifies the verification problem at hand. This does increase the circuit complexity, but only before synthesis! In case of a correct circuit, such an increased circuit should be *redundant*. The described technique relies on the synthesis and SAT tools to prove this redundancy. As clearly demonstrated by the experiments, even if the synthesis is unable to reduce the resulting, inherently redundant circuit to wires/buffer, SAT has a much easier task to prove

TABLE II: Verification run times for bug-free SQRT circuits. #Bits = Radicand bit-width; MO = Memory-out 20GB; TO = Time-out 3600s

# Bits	# Gates	Standard Rewrite (s)	SAT (s)	Simulation (s)	This Work (s)				
					Residue Generation	Re-synthesis	HR-SAT Bug-free	Total Time (Bug-free)	Total Time (Buggy)
6	78	1.93	0.01	0.25	0.01	0.03	0.01	0.05	0.06
12	381	MO	0.01	0.25	0.01	0.03	0.01	0.05	0.06
18	897	MO	0.13	1.9	0.04	0.11	0.01	0.15	0.16
24	1584	MO	2.37	115	0.10	0.24	0.02	0.36	0.38
32	2794	MO	146.9	TO	0.77	1.06	0.02	1.85	1.91
64	10994	MO	TO	TO	1.70	6.07	0.02	7.79	7.85
96	24570	MO	TO	TO	3.78	6.88	0.95	11.61	8.26
128	43522	MO	TO	TO	6.43	10.26	2.92	19.52	21.10
256	263377	MO	TO	TO	9.81	73.53	983.33	1067.3	91.42

the equivalence. In particular, SAT is applied independently to a single-input logic cone, as shown in Figure 5.

We also performed some experiments on buggy circuits by inserting five bugs in random places in the design. The verification results are shown in the last column. Experiments show that for buggy circuits, solving the satisfiability problem (and hence proving the bug) was easier than proving that the functionally correct, bug-free circuit is unSAT. This is not surprising, since in general the unSAT problems are harder to solve (in the worst case the entire solution space may need to be examined). In addition, the solution provided by SAT provides a counter-example that can be used to identify the bug. The debugging is a challenging problem, and it is part of our future work.

The SQRT circuits discussed here are based on a shift/subtract algorithm, similar to dividers. In principle, since our technique does not depend on the internal structure of the circuit (or the algorithm it implements), it should also handle other types of SQRT circuits, such as those based on convergence algorithm, as long as Q_{ref} and R_{ref} can be generated. Addressing these designs is also a part of future work.

The proposed technique can be also applied to other arithmetic circuit with more than one primary output, such as dividers, governed by the characteristic equation $X = Q \cdot D + R$. It is particularly useful for circuits with non-linear polynomial characteristic function. In the future, we will apply this technique to divider circuits, which currently suffer from memory explosion during algebraic rewriting. To the best of our knowledge, this is the first work that was able to successfully verify large integer square-root circuits.

ACKNOWLEDGMENT:
This work has been supported by a grant from the National Science Foundation, award No. CCF-1617708.

REFERENCES

[1] C. Yu, W. Brown, D. Liu, A. Rossi, and M. J. Ciesielski, "Formal verification of arithmetic circuits using function extraction," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2131–2142, 2016.

[2] T. Pruss, P. Kalla, and F. Enescu, "Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases," in *DAC'14*, 2014, pp. 1–6.

[3] J. Harrison, "Formal verification of square root algorithms," in *Formal Methods in Systems Design*, 2003, p. 2003.

[4] C. Yu and M. Ciesielski, "Efficient parallel verification of Galois field multipliers," *ASP-DAC 2017*, 2017.

[5] E. M. Clarke, S. M. German, and X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques," in *(CAV)*. Springer, 1996, pp. 111–122.

[6] R. Kaivola and M. Aagaard, "Divider circuit verification with model checking and theorem proving," *Theorem Proving in Higher Order Logics*, pp. 338–355, 2000.

[7] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Tran. on Comp.*, vol. 100, no. 8, pp. 677–691, 1986.

[8] R. E. Bryant and Y. Chen, "Verification of arithmetic functions with binary moment diagrams," Pittsburgh, PA, USA, Tech. Rep., 1994.

[9] Yimg-An Chen and Bryant, "*phdd: an efficient graph representation for floating point circuit verification," in *1997 Proceedings of IEEE ICCAD*, Nov 1997, pp. 2–7.

[10] E. Pavlenko, M. Wedler, D. Stoffel, and W. Kunz, "Stable: A new QF-BV smt solver for hard verification problems combining Boolean reasoning with computer algebra," in *DATE*, 2011, pp. 155–160.

[11] A. Mahzoon, D. Große, and R. Drechsler, "Polycleaner: Clean your polynomials before backward rewriting to verify million-gate multipliers," in *Proc. International Conference on Computer-Aided Design, ICCAD*, 2018, pp. 129:1–129:8.

[12] M. Ciesielski, T. Su, A. Yasin, and C. Yu, "Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model," *IEEE TCAD*, pp. 1–1, 2019.

[13] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers," in *2019 56th ACM/IEEE DAC*, June 2019, pp. 1–6.

[14] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2017.

[15] M. H. Haghbayan and B. Alizadeh, "A dynamic specification to automatically debug and correct various divider circuits," *INTEGRATION, the VLSI journal*, vol. 53, pp. 100–114, 2016.

[16] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification," *URL http://www.eecs.berkeley.edu/˜ alanmi/abc*, 2007.

[17] P. Behrooz, "Computer arithmetic: Algorithms and hardware designs," *Oxford University Press*, vol. 19, pp. 512 583–512 585, 2000.

[18] I. Koren, *Computer Arithmetic Algorithms*. Universities Press, second edition, 2002.

[19] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Proc. Intl. Conf. on Computer-Aided Verification*, 2010, pp. 24–40.

[20] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, p. 53, 2005.