Functional Verification of Arithmetic Circuits: Survey of Formal Methods

Maciej Ciesielski, Atif Yasin, Jiteshri Dasari
University of Massachusetts, Amherst, MA, USA
ciesiel@umass.edu, atifyasin2@gmail.com, jdasari@umass.edu

***Abstract*—This paper gives a brief survey of current state-of-the-art techniques for formal verification of arithmetic circuits with suggestions for future work. In contrast to standard BDD or SAT-based approach that require a reference circuit it concentrates on Symbolic Computer Algebra (SCA) and related techniques that verify the circuits w.r.t. its abstract arithmetic specification. We examine the original computer algebra method; review the algebraic techniques of forward and backward rewriting; and AIG rewriting. We also propose a "hardware rewriting" method, which replaces algebraic rewriting by hardware synthesis of the circuit under verification appended with an inverse of the circuit, expecting it to be reduced to a redundant one.**

I. Introduction

Verification of arithmetic circuits and datapaths poses a considerable challenge due to its size and large bit-widths of their operands. Strictly Boolean methods, based on BDDs [1] and SAT [2], model the problem as equivalence checking problem, which requires “bit blasting”, flattening of the design into a bit-level netlist. This makes it inefficient for complex circuits such as multipliers. Furthermore, these approaches rely on a trusted reference circuit, which may not be available. The state of the art techniques for arithmetic circuit verification are largely based on Symbolic Computer Algebra (SCA), which represents arithmetic circuits in algebraic domain and model the word-level operands and outputs as polynomials.

There are two basic flavors of these techniques: the classical methods, based on Gröbner basis polynomial reduction [3][4][5][6]; and others, based on a more practical implementation, called *algebraic rewriting* [7]. The goal of this paper is to briefly review the existing techniques in this domain and put them in the context of formal SCA theory. We also describe how these methods can be implemented using efficient modern synthesis tools, such as ABC [8].

The rest of the paper is organized as follows: Section II reviews basic concepts of Symbolic Computer Algebra (SCA) approach. Section III describes the backward and forward *algebraic rewriting* scheme and use of AIG representation to speed up the rewriting process. It also reviews the idea of *spectral method.* Section IV introduces a new concept of *hardware rewriting*, which replaces the expensive algebraic rewriting with hardware synthesis. Finally, Section V discusses applications of the described methods to arithmetic circuit verification and shows some state-of-art results in this field.

II. Computer Algebra Approach

The work in formal verification of arithmetic circuits was pioneered by [9] and [10] who applied concepts from computer algebra and algebraic geometry to formal verification of integer arithmetic circuits and was then extended to Galois Fields [4]. In this approach, the circuit elements and the specification are represented in algebraic domain, as polynomial rings. The circuit components are represented by a set of polynomials G (implementation), and its functionality is given by polynomial F (specification). The verification problem is then formulated as a proof obligation that the implementation (G) satisfies the specification (F) [3][4][5][11].

Mathematically, such a proof is achieved by reducing the specification polynomial F modulo G to a *normal form* and testing if it reduces to zero (i.e., vanishes over $\mathbb{Z}_{2^n}$), known as the *ideal membership test*. The reduction can be accomplished using a computer algebra system, such as Mathematica or Singular [12]. If the result of such a reduction is zero, the circuit satisfies the specification, proving that the circuit is functionally correct. However, in order to trust the result, the set G must satisfy certain properties, given in a canonical form called *Gröbner basis* [13]. In general, computing a complete Gröbner basis is computationally expensive. However, in the case of combinational circuits, with properly ordered set of components (such as logic gates or circuit modules), the resulting set G already constitutes a Gröbner basis (GB). Specifically, if variables in each polynomial of G are ordered from the gate outputs to their inputs, then all leading monomials of the polynomials are relatively prime. This ensures that the corresponding set G automatically constitutes a Gröbner basis, obviating the expensive computation of the complete GB. In gate-level circuits with logic gates the solution must be further restricted to binary variables by imposing additional constraints, $\langle x^2 - x\rangle$, known as *field polynomials*.

III. Algebraic Rewriting

A different approach has been proposed in [7], whereby the expensive polynomial reduction modulo GB has been replaced by a computationally simpler *algebraic rewriting* technique. The circuit is modeled as a network of interconnected logic gates and each gate is modeled as a unique polynomial $f_i[X]$ with binary variables X and coefficients in $\mathbb{Z}_2$. Such a polynomial is also referred to as a *pseudo-Boolean polynomial.* Table I presents algebraic models of some of the basic Boolean operators [7].

TABLE I: Algebraic models of basic logic functions.

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)	$a \oplus b$	$a+b-2ab$
XOR3(a,b,c)	$a \oplus b \oplus c$	$a+b+c-2ab-2ac-2bc+4abc$
MAJ3(a,b,c)	$a \wedge (b \vee c) \vee b \wedge c$	$ab+ac+bc-2abc$

The function computed by an arithmetic circuit is represented as a *specification* polynomial in the primary input variables, denoted F_{spec}. For example, the specification of an n-bit unsigned integer multiplier, $Z = A \cdot B$ with inputs $A = [a_0, \cdots, a_{n-1}]$ and $B = [b_0, \cdots, b_{n-1}]$, is described by $F_{spec} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} a_i b_j$. The result of the computation, stored in the primary output bits, is similarly expressed as a polynomial, called *output signature*, S_{out}. For circuits such as adders and multipliers, such a polynomial is linear, uniquely determined by the m-bit encoding of the output. For example, for an unsigned arithmetic circuit with m output bits, $S_{out} = \sum_{i=0}^{m-1} 2^i z_i$. As we shall see later, S_{out} can be nonlinear for functions such as dividers, SQRT and others.

Algebraic rewriting is the process of transforming output signature S_{out} into a polynomial expressed in primary inputs, the *input signature*, S_{in}, using algebraic models of the internal gates of the circuit, as in Table I. In addition to the gate polynomials shown in the Table, each Boolean variable x is represented by a *field polynomial* $< x^2 - x >$. The zeros of this polynomial (0 or 1) impose the desired Boolean value on the variable.

In a functionally correct circuit, the resulting S_{in} should match the specification polynomial, F_{spec}. If the specification is not known, the resulting S_{in} provides the arithmetic function computed by the circuit. Hence, the method can be used as a reverse engineering tool for arithmetic function extraction.

In Computer Algebra approach, adopted in [3][5][6] the specification F of the circuit is defined as the difference between the output polynomial and the actual functional *specification* in the primary inputs; in our terminology, $F = S_{out} - F_{spec}$. The verification goal is then to prove that $F \ mod \ G$ reduces to zero, where G is the set of implementation polynomials. In the approach based on algebraic rewriting, it is the output signature S_{out} that is being reduced modulo G to the input signature S_{in}, instead of reducing F to 0. In the correct circuit, the resulting S_{in} is expected to match the specification polynomial, F_{spec}. That is, we need to check if $S_{out} \xrightarrow{G}_{+} S_{in} = F_{spec}$.

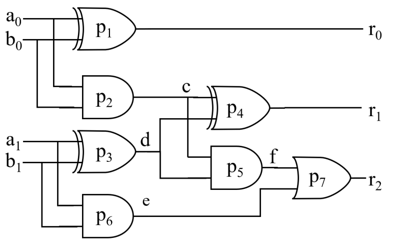
The reduction can be actually performed in both directions, either by *forward rewriting* or *backward rewriting*. In computer algebra approach this is equivalent to the reduction using topological (forward) term order vs reverse topological (backward) order.

The work of [6] revisits the algebraic rewriting techniques from [7] and [5] and provides the proof of correctness for the underlying rewriting and its equivalence to the Symbolic Computer Algebra (SCA) approach. Specifically, the paper justifies the use of the theory of ideal membership (in principle applicable to $\mathbb{Q}[X]$) to prove properties of integer arithmetic circuits. It points out that, since the leading coefficients of the gate polynomials forming the Gröbner basis are +1 or -1, polynomial reduction never introduces fractional coefficients and their computation remains in $\mathbb{Z}$. Hence the "dedicated implementations in [7] and [5] can rely on computation in $\mathbb{Z}$ only, while remaining sound and complete" [11].

The remainder of this section compares two types of algebraic rewriting: backward rewriting and forward rewriting, emphasizing their relationship to Gröbner basis reduction.

A. Backward Rewriting

In the example of Figure 1 the output signature of the circuit is $S_{out} = 4r_2 + 2r_1 + r_0$. It is rewritten into an input signature S_{in} to be compared with the circuit specification $F_{spec} = 2a_1 + 2b_1 + a_0 + b_0$. The following backward rewriting steps are



$$\begin{aligned}
p_1 &= r_0 - (a_0 + b_0 - 2a_0b_0)\\
p_2 &= c - a_0b_0\\
p_3 &= d - (a_1 + b_1 - 2a_1b_1)\\
p_4 &= r_1 - (c + d - 2cd)\\
p_5 &= f - cd\\
p_6 &= e - a_1b_1\\
p_7 &= r_2 - (e + f - ef)
\end{aligned} \quad (1)$$

Fig. 1: A two-bit adder circuit and gate polynomials G.

applied starting with $S_{out} = 4r_2 + 2r_1 + r_0$ and ending at $S_{in} = 2a_1 + 2b_1 + a_0 + b_0$. In the following, F/p denotes the division of polynomial F by the gate polynomial p. The subsequent divisions by next polynomials are denoted as $F/...p$

$$\begin{aligned}
&F = S_{out} = 4r_2 + 2r_1 + r_0\\
&F/p_7 = 4e + 4f - 4ef + 2r_1 + r_0\\
&F/...p_4 = 4e + 4f - 4ef + 2c + 2d - 4cd + r_0\\
&F/...p_5 = 4e - 4edc + 2c + 2d + r_0\\
&F/...p_1 = 4e - 4edc + 2c + 2d + a_0 + b_0 - 2a_0b_0\\
&F/...p_2 = 4e - 4eda_0b_0 + 2d + a_0 + b_0\\
&F/...p_3 = 4e - 4ea_0b_0(a_1 + b_1 - 2a_1b_1) + 2(a_1 + b_1 - 2a_1b_1) + a_0 + b_0\\
&F/...p_6 = 4a_1b_1 - 4a_1b_1a_0b_0(a_1 + b_1 - 2a_1b_1) + 2(a_1 + b_1) - 4a_1b_1\\
&\qquad + a_0 + b_0 = 2a_1 + 2b_1 + a_0 + b_0\\
&F/(G, J_0) = S_{in} = 2a_1 + 2b_1 + a_0 + b_0
\end{aligned} \quad (2)$$

Note that the computed S_{in} matches the expected specification F_{spec}, proving that the circuit correctly implements a two-bit adder.

A number of rewriting ordering strategies are employed to enable cancellation of terms as early as possible and significantly reduce the size of the intermediate polynomials. They include keeping the polynomials of gates with same inputs adjacent and recognizing "vanishing monomials" (see Section V, [14]). The most important cancellations occur due to the reduction of monomials x^2 to x, whenever a nonlinear term x^2 is generated during the rewriting. This is more efficient than performing division by polynomials $< x^2 - x >$, associated with internal variables x, as done by the traditional SCA reduction.

B. Forward Rewriting

In this variant of algebraic rewriting the base polynomials G representing the gates are ordered in *topological order*, from gate inputs to outputs. In our two-bit adder example the forward term order is $\{a_0, b_0, a_1, b_1\} > \{c, d, \} > \{e, f\} > \{r_0, r_1, r_2\}$ The function to be reduced is

$$F = (a_0 + b_0 + 2a_1 + 2b_1) - (r_0 + 2r_1 + 4r_2)$$

and the corresponding implementation base G is:

$$\begin{aligned}
p_1 &= -2a_0b_0 + a_0 + b_0 - r_0 \\
p_2 &= a_0b_0 - c \\
p_3 &= -2a_1b_1 + a_1 + b_1 - d \\
p_4 &= -2cd + c + d - r_1 \\
p_5 &= cd - f \\
p_6 &= a_1b_1 - e \\
p_7 &= -ef + e + f - r_2
\end{aligned} \tag{3}$$

In this case, the *input* variables of the gates are among the *leading* terms of the polynomial, and the gate output is in the tail. Set G under such a term order does not form Gröbner basis, since the same variable may appear more than once as a leading term (primary inputs and signals with fanout); and the leading terms are not relatively prime. In this case, a Gröbner basis must be computed to effect this reduction. In addition, the set of field polynomials $J_0 =< x^2 - x >$ needs to be added to the base for all the circuit variables x, which further adds to the complexity.

The effect of the reduction, obtained by dividing the current polynomial by the leading terms of the elements of G is illustrated by the following sequence.

$$\begin{aligned}
&F = a_0 + b_0 + 2a_1 + 2b_1 - r_0 - 2r_1 - 4r_2 \\
&F/(a_0 + b_0) = 2a_0b_0 + 2a_1 + 2b_1 - 2r_1 - 4r_2 \\
&F/...(a_0b_0) = 2c + 2a_1 + 2b_1 - 2r_1 - 4r_2 \\
&F/....(a_1 + b_1) := 2c + 2d + 4a_1b_1 - 2r_1 - 4r_2 \\
&F/....(c + d) = 4cd + 4a_1b_1 - 4r_2 \\
&F/....(cd) = 4f + 4a_1b_1 - 4r_2 \\
&F/....(a_1b_1) = 4f + 4e - 4r_2 \\
&F/....(e + f) = 4ef
\end{aligned} \tag{4}$$

The result of this reduction is a non-empty remainder, $R = 4ef$, which cannot be reduced further, neither with G nor with J_0. For example, reduction over $< e^2 - e >$ would produce $R = 4e^2f^2 = 4ef = 4e^2f^2$, etc., and similarly with reduction with p_7, creating an infinite loop. Furthermore, since G is not a Gröbner basis, the result cannot be trusted, while it is possible that the circuit is correct. To confirm this, one needs to check if $R = 4ef$ can be reduced to 0 by other means. This, in fact, can be achieved using backward rewriting with base G. The reduction is shown below.

$$\begin{aligned}
R' &= ef = (a_1b_1)(dc) \\
&= (a_1b_1)(a_1 + b_1 - 2a_1b_1(a_0b_0)) \\
&= (a_1b_1)(a_1a_0b_0 + b_1a_0b_0) - 2a_1b_1a_0b_0 \\
&= (a_1b_1a_0b_0 + a_1b_1b_1a_0b_0 - 2a_1b_1a_1b_1a_0b_0) \\
&= (a_1b_1a_0b_0 + a_1b_1a_0b_0 - 2a_1b_1a_0b_0) = \ 0
\end{aligned} \tag{5}$$

This successful reduction uses the relationship $a_1^2 = a_1$ and $b_1^2 = b_1$, implied by the field polynomials $J_0 =< a_1^2 - a_1; b_1^2 - b_1 >$. In summary, in order to solve this problem, one can either prove the residual expression to be a zero function by backward rewriting, or create a complete Gröbner basis $B =< G,\ J_0 >$ and use it to effect the reduction. Both solutions are computationally expensive and not scalable. In the case of a faulty circuit, the residual reduction will not be able to reduce R to zero, or, alternatively, the constructed Gröbner basis B will not be able to reduce F to 0, indicating that the circuit is faulty.

In conclusion, forward reduction is not complete as it may not terminate with a conclusive reduction. However, it can be useful in *debugging*; under certain conditions propagation of the input signature to the outputs can reveal the location of a bug [15], [16].

C. AIG Rewriting

The process of algebraic rewriting can be significantly improved by using a functional, And-Invert Graph (AIG) representation employed in ABC tool [8]. ABC uses the cut enumeration to detect the XOR and Majority (MAJ) functions with a common set of variables. Those nodes are essential in identifying half-adders (HA) and full-adders (FA), the basic components of an arithmetic circuit. AIG rewriting then skips over significant portions of the circuitry, from the inputs to the outputs of the adders, as shown in Figure 2(b), significantly speeding up the rewriting process.

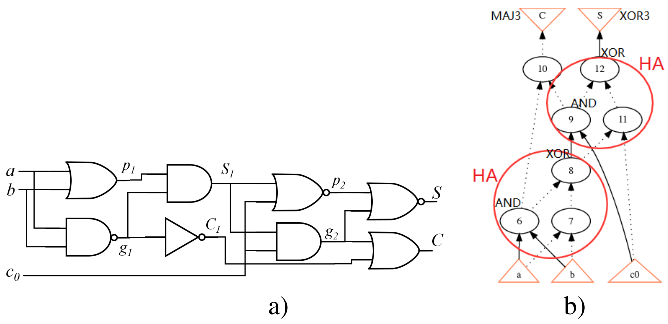


Fig. 2: Full Adder: a) circuit diagram; b) AIG representation

In fact, the AIG data-structure makes it possible to directly propagate the coefficients of the polynomials, without actually performing alphanumeric string manipulation and term substitution. This gives a rise to the *spectral* method, [17], which has been integrated with ABC as command "$\&poly - w$".

Consider an n-bit integer multiplication scheme, shown in Figure 3(a) for $n = 4$. The ovals represent partial product terms that are added column-wise for each bit of the result. Let i, be a bit position of the result, and N_i be the number of product terms added at this bit position. It can be shown that the graph of $N(i)$ uniquely determines the type of the arithmetic function implemented by the circuit, regardless of its hardware representation. We refer to such a graph as the *spectrum* of the circuit. Figure 3(b) shows the spectrum for a 4-bit multiplier. Spectra for 2-operand and 3-operand multipliers are shown in Figure 4 a) and b), respectively. Algebraic spectrum for an n-bit adder will be a flat line, since the number of terms is the same for each bit. The shape of

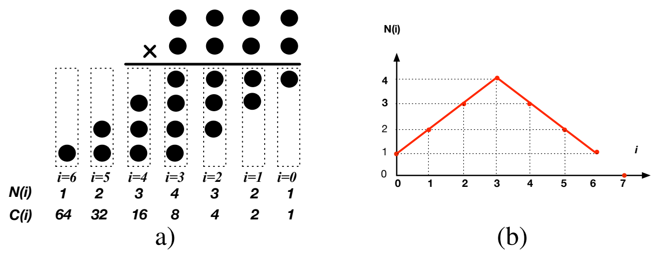
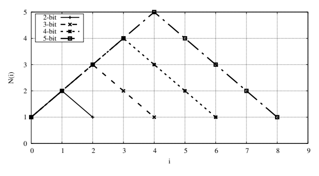


Fig. 3: Spectrum of a four-bit Multiplier.

the spectrum remains the same for a given arithmetic function and can be used to determine its type. The spectrum of a given circuit does not depend on its hardware implementation or the internal structure but only on the arithmetic function it implements. Thanks to this important feature it can be used to determine the type of arithmetic operation performed by the circuit. Similar formulas and spectra can be derived for


(a) $F = A \cdot B$.

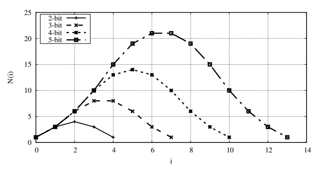

(b) $F = A \cdot B \cdot C$.

Fig. 4: Spectra for multipliers for { 2,3,4,5 } bit-widths.

other datapath operators, such as MAC, fused multiply/add operation, and others[1].

Note that in a monolithic arithmetic function (i.e., function composed of only one arithmetic operator) each monomial in its polynomial representation contains the same number of variables. However, a fused multiplier $A + B \cdot C = \sum 2^i a_i + \sum 2^{j+k}(b_j c_k)$ will contain both single-variable terms $\{a_i\}$ and two-variable terms $\{b_j c_k\}$. In this case, the circuit is represented by two independent spectra, rather than a single one. Even though the spectrum can be derived from the polynomial expression of the input signature Sig_{in}, it can be computed without algebraic rewriting, using only the AIG representation, as mentioned in the preceding section.

IV. Hardware Rewriting

This section describes an original method for verifying arithmetic circuits, in which algebraic rewriting is replaced by hardware synthesis. This approach has been motivated by a need to verify integer and fractional dividers, essential components of fixed and floating point datapaths. In contrast to other arithmetic circuits, such as adders or multipliers, divider does not have a closed form formula that could its express its output as a function of the inputs. Instead, its functionality is governed by the following equation:

$$X = D \cdot Q + R, \quad \text{with } R < D \tag{6}$$

[1]A larger set of algebraic spectra are available in our online spectrum gallery: https://ycunxi.github.io/cunxiyu/spectrum_gallery.html

where X is the dividend, D the divisor, and Q, R are the quotient and remainder, respectively.

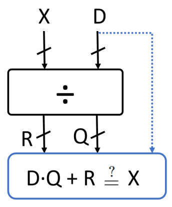


Fig. 5: Divider verification model.

Figure 5 shows an abstract model that captures this expression and suggest the way to solve the verification problem. The upper part of the diagram is the divider under verification. The lower box "reverses" the division X/D by computing $Q \cdot D + R$ from the quotient Q and the remainder R, produced by the divider. The goal is to prove that the computed result matches the original dividend X, i.e. $Q \cdot D + R = X$. Satisfying the condition $R < D$ is a separate problem, discussed in [18].

One possible way to solve this problem is to create a circuit $Y = Q \cdot D + R$ and check the equivalence between its output Y and the dividend X. This can be handled by a standard SAT technique: create a miter between the dividend input X and output $Y = QD + R$ of the circuit and check if the CNF formula of the resulting miter circuit is unsatisfiable (unSAT). Unfortunately, the dividers greater than 16 bits could not be verified using this method. Similarly, an attempt to affect algebraic rewriting of such a structure, by transforming $Q \cdot D + R$ into the dividend X, was not successful: for circuits with dividends greater than 10 bits the size of intermediate polynomials becomes prohibitively large, causing memory blowup.

To address this problem, a *layered* approach, originally proposed in [18], can be used. Verification is applied to each row (layer) of the divider, representing one step of the controlled subtraction of the shifted dividend, and producing a quotient bit q_i. The rewriting can then be applied to one layer at a time. This approach is justified by noting that logic between two adjacent rows of the divider is not optimized during synthesis and the partial remainder signals R_i are preserved during synthesis. This has been confirmed by [19], Theorem 2, which states that "In an array divider, there is no way to optimize carry logics (CL) of adjacent rows i, $i + 1$; no optimization technique is able to combine $CL_{i,j}$ and $CL_{(i+1),j}$". The circuit can be synthesized horizontally along each layer, use different types of adders, speed up carry propagation, etc., but not optimized vertically across their boundaries. This process can also be done in a speculative, parallel manner, since the form of each polynomial at the row boundary is known, and can be stopped when one of the layers does not produce the expected result. This way the source of an error is constrained to a particular layer and the propagation of rewriting will stop there to examine the bug.

Unfortunately, such a layered algebraic rewriting is still non-scalable for circuits with dividends beyond 21 bits. And this bring us to the idea of *hardware rewriting*, originally proposed in [20] for SQRT circuits.

The main idea is similar to that shown in Figure 5 but applied to a single layer i, which computes the quotient bit q_i and the partial remainder R_i from the divisor D and the previous remainder R_{i-1}. The output Z_i of such an inverse circuit simply computes $Z_i = D \cdot q_i + R_i = R_{i-1}$. The goal is to prove that the n-bit output Z_i matches the n-bit input R_{i-1} bit-by bit. This is accomplished by synthesizing the circuit and checking the equivalence between its input and output bits. Our initial experiments show that for layers up to 24-bit wide, the result is a redundant circuit composed of a set of direct wires/buffers, connecting the bits of Z_i with bits of R_{i-1}. For larger circuits, the synthesis does not reduce the structure to such a redundant state, but it can be trivially verified using bit-by bit SAT, or by a simple XOR comparison. The verification of the entire circuit is accomplished by *composing* the verification results of individual layers.

The technique of layered verification shows to be efficient and scalable and can verify dividers of up to 255-bit dividends in just 123 seconds, as shown in Section V. This is significantly faster than SBIF verification of the entire divider at once (990 sec). This is in contrast to SAT approach, which times out after one hour on a single 32-bit layer of the divider.

V. Applications and Results

The verification techniques described in this paper have been successfully applied to a number of large circuits, ranging from integer and Galois field multipliers to dividers and sqrt circuits. The following is a brief summary of the main achievements in the integer arithmetic circuits. The summary of these results, compiled from different sources, are shown in Table II. They are not intended for direct comparison but provided just to give a brief idea of the order of execution time to solve some of these problems. The CPU time ranges considerably depending on the circuit architectures, level of optimization/synthesis, platform used, etc.

ARTi / BitFlow [7][21]: The initial results in multiplier verification considered basic array architectures, including standard and Booth multiplier with several types of adder trees. These tools are capable of solving the verification problem for multiplier circuits up to 256-bit operands in 285 seconds.

RevSCA 2.0 [14]: An advanced SCA-based technique to solve large and non-trivial multipliers. It combines reverse engineering and removal of local vanishing monomials whose origin are the gates where both outputs of Half Adders converge at an AND gate. The efficiency of this approach has been demonstrated using an extensive set of multipliers of different architectures, including different product term generation (simple and Booth), different adder trees, and different final adders. As an example, they report that a 512-bit multiplier contained 38 million vanishing monomials, whose removal saved the process from memory explosion.

AMulet 2.0 [22]: This work describes a fully automated tool, Amulet-2.0. The tool detects the final stage of the adder, replaces it with a trusted ripple-carry adder and proves the modified structure using SCA and SAT. The experiments were performed on simple multipliers up to the input size 2,048 using AOKI benchmark set [23]. On this set AMulet 2.0 outperforms other tools and is an order of magnitude faster on large multiplier circuits. This is the only tool known to us that also provides a proof *certification*. In the certification mode, AMulet 2.0 tracks polynomial operations in the selected proof format and prints out the gate constraints, the generated proof, and the specification to the provided files.

SBIF & DCs [24]: This work concentrates on dividers using SCA/rewriting approach. It recognizes the failure of a straightforward application of rewriting caused by the excessive number of generated monomials. The reason of failure ican be summarized as follows:
1) There is a large number of equivalent and antivalent signals present in the circuit that converge causing many monomials to vanish; these should be detected and removed as early as possible. To identify those monomials, the circuit is simulated with input vectors satisfying the input range constraint, $0 \leq X \leq D \cdot 2^{n-1}$, where n is the size of the divider D. The signals that fall in the same equivalence class are then checked using SAT to be classified as equi- or anti-valent. This is supported by signal propagation from the primary inputs, hence termed as "SAT-based Information Forwarding" (SBIF). The affected monomials are removed before they get propagated further to avoid potential memory blowout.
2) Another important source of the problem is that practical divider circuits are optimized based on the input constraint, $0 \leq X \leq D \cdot 2^{n-1}$. In the resulting architecture the MSB cells in the upper levels of the divider have unused outputs, making them "unclean" for backward rewriting. This deprives the rewriting process of the monomials that are essential to annihilate monomials from other outputs generated on the same level. Because of this incomplete data, no reduction in polynomial size is possible, causing an uncontrolled increase in the polynomial size. These redundant polynomials are analogous to don't cares (DC) in logic circuits and are modeled as *satisfiability don't cares*.

Instead of computing all don't cares, the paper proposes a method to choose a subset of the DCs that will minimize the polynomial size at that level. To support this method, they extract atomic blocks (half and full adders) and use BDDs and standard Boolean logic techniques to compute satisfiability DCs at the inputs to these blocks. When the size of the polynomial increases by some predefined rate, they stop the procedure, backtrack to the previous step, apply the ILP optimization to reduce the vanishing monomials, and continue with polynomial rewriting. This approach proved to be efficient in restricting the size of the intermediate polynomials. It can handle non-restoring 256-bit and 512-bit dividers in 16.5 min and 162 minutes, respectively. One must note, however, that this method depends on the extraction of the HA/FA blocks to gain access to the their inputs on which to apply don't cares, which is another form of the layering approach advocated in [18].

HW-Rewrite [20]: Our (yet unpublished) recent work on divider verification applies hardware rewriting with a layered approach but solves the problem faster. The 256-bit restoring divider, considered to be a more complex and harder to verify than the non-restoring divider considered in [24], takes only 123 sec to verify for all layers. The CPU time can be two orders of magnitude shorter if the layers are verified concurrently (0.5 second per layer). A disadvantage of this approach is that it needs to explicitly know the boundaries between the layers.

Theorem Provers [25]: There has been much work on formal verification of arithmetic circuits using Theorem Provers, such as ACL2. They use term rewriting techniques to prove the correctness of a wide variety of multiplier architectures. Although relying on knowledge of the design hierarchy, their CPU runtimes are competitive with other verification methods described here. In particular, they have been quite successful in proving large 1024-bit wide multipliers.

Complexity: We conclude this paper with the analysis of computational and space complexity and theoretical bounds in arithmetic circuit verification, described in [26]. This work examines a slew of different approaches (BDD, BMD, SAT, SCA, and algebraic rewriting), and concludes that the verification process for integer arithmetic circuits requires at most linear space and quadratic time w.r.t. the size of the circuit. However, as clearly noted there, this result applies only to "clean" circuits with a regular structures. Circuits with more advanced and optimized structures, such as carry look-ahead adders and Booth multipliers are not considered there. This is in line with a large body of experimental results, which shows the difficulty and higher computational complexity in verifying highly synthesized circuits.

TABLE II: Results compiled from different sources.

Team	Circuit size (bits)	Circuit type	CPU time (sec)
BitFlow/ARTi	128	Mult synth	400
[21][7]	256	Mult clean	285
RevSCA	256	Multipliers	2,500-6,100
[14]	512	Multipliers	48,700-115,500
AMulet	1024	Mult clean	300
[22]	2048	Mult clean	3,000
SBiF/DC	256	Divider	990
[24]	512	Divider	9,669
HW-Rewrite	256	Divider	123
[20]	256	SQRT	91
ACL2	1024	Multiplier	220
[25]	1024	Multiplier	300

ACKNOWLEDGMENT:

This work has been supported by a grant from the National Science Foundation, Award No. CCF-2006465.

References

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

[2] M. Ganai and A. Gupta, *SAT-based scalable formal verification solutions*. Springer, 2007.

[3] 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.

[4] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, September 2013.

[5] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining Gröbner basis with logic reduction," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1048–1053.

[6] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1556–1561.

[7] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2131–2142, 2016.

[8] A. Mishchenko *et al.*, "ABC: A system for sequential synthesis and verification," *URL http://www. eecs. berkeley. edu/˜ alanmi/abc*, 2007.

[9] N. Shekhar, P. Kalla, and F. Enescu, "Equivalence verification of polynomial data-paths using ideal membership testing," *TCAD*, vol. 26, no. 7, pp. 1320–1330, July 2007.

[10] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G.-M. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," *CAV*, pp. 473–486, July 2008.

[11] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD'17*, 2017.

[12] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, "SINGULAR 3-1-6 A Computer algebra system for polynomial computations," Tech. Rep., 2012, http://www.singular.uni-kl.de.

[13] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms*. Springer, 1997.

[14] A. Mahzoon, D. Große, and R. Drechsler, "REVSCA-2.0: SCA-based formal verification of non-trivial multipliers using reverse engineering and local vanishing removal," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[15] S. Ghandali, C. Yu, D. Liu, W. Brown, and M. Ciesielski, "Logic debugging of arithmetic circuits," in *ISVLSI'15*, July 2015.

[16] T. Su, A. Yasin, C. Yu, and M. Ciesielski, "Computer algebraic approach to verification and debugging of Galois field multipliers," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5.

[17] C. Yu, T. Su, A. Yasin, and M. Ciesielski, "Spectral approach to verifying non-linear arithmetic circuits," *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 261–267, 2019.

[18] A. Yasin, T. Su, S. Pillement, and M. Ciesielski, "Functional verification of hardware dividers using algebraic model," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2019, pp. 257–262.

[19] 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.

[20] A. Yasin, T. Su, S. Pillement, and M. Ciesielski, "SPEAR: hardware-based implicit rewriting for square-root circuit verification," *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 532–537, 2020.

[21] M. Ciesielski, T. Su, A. Yasin, and C. Yu, "Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model," *IEEE TCAD*, vol. 39, no. 6, pp. 1346–1357, 2019.

[22] D. Kaufmann and A. Biere, "AMulet 2.0 for verifying multiplier circuits." in *TACAS (2)*, 2021, pp. 357–364.

[23] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi, "Formal design of arithmetic circuits based on arithmetic description language," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 89, no. 12, pp. 3500–3509, 2006.

[24] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler, "Verifying dividers using symbolic computer algebra and don't care optimization," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1110–1115.

[25] M. Temel and W. A. Hunt, "Sound and automated verification of real-world RTL multipliers," in *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2021, pp. 53–62.

[26] M. Barhoush, A. Mahzoon, and R. Drechsler, "Polynomial word-level verification of arithmetic circuits," in *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*, 2021, pp. 1–9.