Fast Algebraic Rewriting Based on And-Inverter Graphs

Cunxi Yu, *Student Member, IEEE*, Maciej Ciesielski, *Senior Member, IEEE*, and Alan Mishchenko, *Senior Member, IEEE*

***Abstract*—Constructing algebraic polynomials using computer algebra techniques is believed to be state-of-the-art in analyzing gate-level arithmetic circuits. However, the existing approach applies algebraic rewriting directly to the gate-level netlist, which has potential memory explosion problem. This paper introduces an algebraic rewriting technique based on the and-inverter graph (AIG) representation of gate-level designs. Using AIG-based cut-enumeration and truth table computation, an efficient order of algebraic rewriting is identified, resulting in dramatic simplifications of the polynomial under construction. An automatic approach, which further reduces the complexity of algebraic rewriting by handling redundant polynomials, is also proposed.**

***Index Terms*—And-invert graphs (AIGs), computer algebra, computer arithmetic, formal verification.**

I. Introduction

Importance of arithmetic verification problem grows with an increased use of arithmetic modules in embedded systems to perform computation-intensive tasks in multimedia, signal processing, and cryptography applications. One of the remaining challenges in formal verification is formal verification of gate-level integer arithmetic circuits, such as multipliers, used extensively in those applications. Despite a considerable progress in verification of random and control logic, advances in formal verification of arithmetic designs have been slow. This can be attributed to the difficulty in the efficient modeling of arithmetic circuits and datapaths without resorting to computationally expensive Boolean methods, such as binary decision diagram (BDD), satisfiability, satisfiability modulo theories (SMT), etc., that require "bit blasting," i.e., flattening the design to a bit-level netlist. However, recently, formal techniques based on *computer algebra* have been successfully applied to the verification problems of gate-level arithmetic circuits.

Computer algebra techniques, which construct the polynomial representation of a gate-level arithmetic circuit, are believed to offer best solution for analyzing arithmetic circuits [1]–[5]. These works address the verification problems of Galois field arithmetic [2] and integer arithmetic implementations, including abstractions and reverse engineering [1], [3]–[5]. The verification problem is typically formulated as a proof that the implementation satisfies the specification, which is solved by polynomial division or algebraic rewriting. The results show that the computer algebra techniques provide several orders of magnitude in performance improvement. The main advantage of computer algebra method in verifying arithmetic circuits is that it provides a large number of polynomial reductions by eliminating nonlinear terms. However, the size of those terms could explode exponentially by rewriting its variables if they were not eliminated at the right time (order).

The order of rewriting or, equivalently, performing polynomial divisions has a significant impact on the performance of the computer algebra techniques [5], [6]. However, these techniques may fail to find efficient rewriting order if they are applied directly to the gate-level netlist. Yu *et al.* [6] compared the performance of algebraic methods of combinational gate-level multipliers when different topological orders are used. It showed that an efficient topological order may not exist in the post-synthesized gate-level netlist. Even if such an order exists, it may be difficult to be identified because of the polynomial reductions hidden in the complex standard cells. In addition, redundant polynomials detected from combinational and sequential arithmetic circuits can provide significant polynomial reductions [7]. However, detecting such polynomials is limited by manual operations and depends on the structure of the circuits.

The approach presented in this paper aims at improving the efficiency of algebraic rewriting in the context of arithmetic verification. It addresses the problem by using a compact and uniform representation of the Boolean network called the and-inverter graph (AIG) [8]. Instead of directly applying algebraic rewriting to the gate-level netlist, it is applied to an AIG. In addition, this approach allows one to automatically generate redundant polynomials, which significantly reduce the complexity of algebraic rewriting.

II. Background

A. Formal Verification of Arithmetic Circuits

Verification of arithmetic circuits is performed using a variation of combinational equivalence checking (CEC) referred to as arithmetic CEC (ACEC) [5]. Several approaches have been applied to equivalence check an arithmetic circuit against its functional specification, including *canonical diagrams*, *satisfiability* (SAT) theories, *theorem proving*, etc. Different variants of canonical, graph-based representations have been proposed, including BDDs, binary moment diagrams [9], Taylor expansion diagrams [10], and other hybrid diagrams. While BDDs have been used extensively in logic synthesis, their application to verification of arithmetic circuits is limited by the prohibitively high memory requirements for complex arithmetic circuits, such as multipliers. Boolean SAT and SMT solvers have also been applied to solve ACEC problems [11]. Recently, several state-of-the-art SAT and SMT solvers have been applied to those problems, including MiniSAT [12], Lingeling [13], Boolector [14], Z3 [15], etc. However, the complexity of checking equivalence of large arithmetic circuits is extremely high [6], [16]. Alternatively, the problem can be modeled as checking equivalence against the arithmetic function, e.g., checking whether the binary encoded output function is equivalent to the expected arithmetic function using bit-vector formulation of SMT. However, the complexity of this method is the same as the CEC method [6].

Manuscript received February 18, 2017; revised June 2, 2017 and October 2, 2017; accepted November 1, 2017. Date of publication November 13, 2017; date of current version August 20, 2018. This work was supported by the National Science Foundation under Grant CCF-1319496 and Grant CCF-1617708. The work of A. Mishchenko was supported by NSA grant Enhanced Equivalence Checking in Crypto-Analytic Applications. This paper was recommended by Associate Editor J. Cortadella. *(Corresponding author: Cunxi Yu.)*

C. Yu and M. Ciesielski are with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003 USA (e-mail: ycunxi@umass.edu; xiangyuzhang@umass.edu; ciesiel@ecs.umass.edu).

A. Mishchenko is with EECS Department, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: alanmi@berkeley.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2017.2772854

0278-0070 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

B. Computer Algebra Approaches

In computer algebra approach, the verification problem is typically formulated as a proof that the implementation satisfies the specification [1]–[5]. This task is accomplished by performing a series of divisions of the specification polynomial by a set of polynomials, representing components that implement the circuit. Techniques based on *Gröbner Basis* demonstrate that this approach can efficiently transform the verification problem into *membership testing* of the specification polynomial in the ideals [2], [4]. All of these works impose a lex term order on the implementation polynomials by performing reverse topological traversal of the circuit, which automatically renders the set of implementation polynomials a Groebner basis. Some of Lv *et al.* [2] and Farahmandi and Alizadeh [4] used Gaussian elimination, rather than explicit polynomial division, to speed up the reduction process. To reduce the number of polynomials, [4] precomputes polynomials corresponding to fanout-free logic cones. A different approach to arithmetic verification of gate-level circuits has been proposed using the algebraic rewriting technique, which transforms the polynomial at the primary outputs to a polynomial in terms of primary inputs (PIs) [1], called *function extraction*. This approach has successfully been applied to 512-bit multipliers, due to a large number of polynomial reductions gained by rewriting a binary encoded polynomial of the outputs [6]. A similar approach has been applied to arithmetic CEC [5], with a novel approach of detecting and eliminating vanishing monomials, i.e., products of variables which evaluate to zero due to reconvergent fanout. Although those works showed good performance in solving arithmetic verification problems, they still suffer from potential polynomial (memory) explosion problem since they are applied to the original gate-level netlist.

C. Boolean Network

Boolean network is a directed acyclic graph with nodes representing logic gates and directed edges representing wires connecting the gates. AIG is a combinational Boolean network composed of two-input AND-gates and inverters [8]. In an AIG, each node has at most two incoming edges. A node with no incoming edges is a PI. Primary outputs are represented using special output nodes. Each internal node in the AIG represents a two-input AND function. Based on DeMorgan's rule, the combinational logic of an arbitrary Boolean network can be transformed into an AIG [17], with the properly labeled edges to indicate the inversion of the signals. AIGs have been extensively used in logic synthesis, technology mapping [17] and formal verification [18].

AIGs have been used to detect unobserved Boolean functions such as *multiplexer* function in an arbitrary gate-level circuits. This method is based on computing a *Cut* in the AIG. A cut C of node n is a set of nodes of the network called *leaves*, such that each path from PIs to n passes through the leaf nodes. Node n is the *root* of a *Cut*. A *Cut* is K-feasible if the number of leaves does not exceed K. The cut function is the function of node n in terms of the cut leaves. An AIG node n in an AIG structure that represents a Boolean function F, is called an F-node. Each node is an AND function and the edges indicate the inversions of Boolean signals.[1] An example of identifying XOR functions embedded in the AIG is shown in Fig. 1. The AIG shown in Fig. 1(b) represents a sub-circuit described in Fig. 1(a). It includes a three-feasible *Cut* of node 9 and a two-feasible *Cut* of node 6, among other possible three-feasible cuts. Let the function of the AIG node with index x be i_x. The function of node 6 is $i_1 \oplus i_2$, and the function of node 9 is $i_1 \oplus i_2 \oplus i_3$. Hence, node 6 is an XOR2-node, and node 9 is an XOR3-node. This means that an embedded XOR3 function consisting of two XOR2s exists and can be detected in the sub-circuit shown in Fig. 1(a). Similarly, an AIG can be applied to identify embedded *majority* functions.

[1]In Fig. 1, the dash edges are inversion signals, e.g., $i_4 = \overline{i_1}\ \overline{i_2}$, $i_5 = i_1 i_2$.

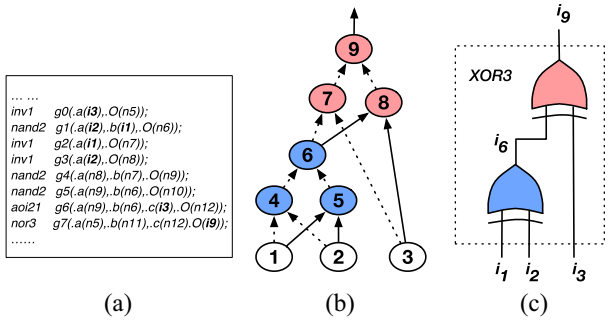

Fig. 1. XOR3 design. (a) Post-synthesized gate-level netlist. (b) AIG of the synthesized netlist. (c) Extracted two XOR2 functions (nodes 6 and 9) and one XOR3 function (node 9).

TABLE I
BOOLEAN AND ALGEBRAIC MODELS OF INV, AND, XOR, AND MAJORITY FUNCTIONS

Operation	Boolean Model	Algebraic Model
INV(a)	$\bar{a}$	1-a
AND(a,b)	a $\wedge$ b	ab
XOR(a,b,c)	a $\oplus$ b $\oplus$ c	a+b+c-2ab-2ac-2bc+4abc
Majority(a,b,c)	(a $\vee$ b) $\wedge$ (a $\vee$ c)	ab+ac+bc-2abc

D. Computer Algebraic Model

In this approach, the circuit is modeled as an AIG containing the following gates: INV, AND, embedded MAJ3, and embedded XOR3. This is in contrast to using a standard-cell network model after synthesis and technology mapping [1]. The following algebraic equations, Table I describe the algebraic model used in this paper.

Similarly to [1], the algebraic rewriting for a circuit is based on two polynomials, referred to as *output signature* and *input signature*. The *input signature*, Sig_{in}, is a polynomial in terms of PI variables that uniquely represents the integer function computed by the circuit, i.e., its *specification*. For example, an n-bit binary adder with inputs $\{a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}\}$, is described by $\text{Sig}_{\text{in}} = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i$. In our approach, the input specification need not be known; it will be derived from the circuit implementation as part of the verification process. The *output signature*, Sig_{out}, of the circuit is a polynomial in terms of the primary output variables. Such a polynomial is uniquely determined by the n-bit encoding of the output, provided by the designer. This means that the binary encoding of the primary output variables is assumed to be known.

E. Simplified Polynomial Construction

According to [6], efficiency of algebraic rewriting of Sig_{out} is determined by the amount of simplifications during polynomial construction. This is because there is a large number of nonlinear terms generated by *carry-out* (MAJ) and *sum*(XOR) functions, since multiplication is affected by a series of additions. Finding the maximum polynomial cancellations has been previously addressed by improving the topological order of the gates [6]. For example, let a sub-polynomial expression be $x_1 + 2x_2 + \ldots$, where $x_1 = \text{XOR3}(a, b, c)$, $x_2 = \text{MAJ3}(a, b, c)$, where *a, b, c* are the inputs of XOR3 and MAJ3 functions. According to equations in Table I,

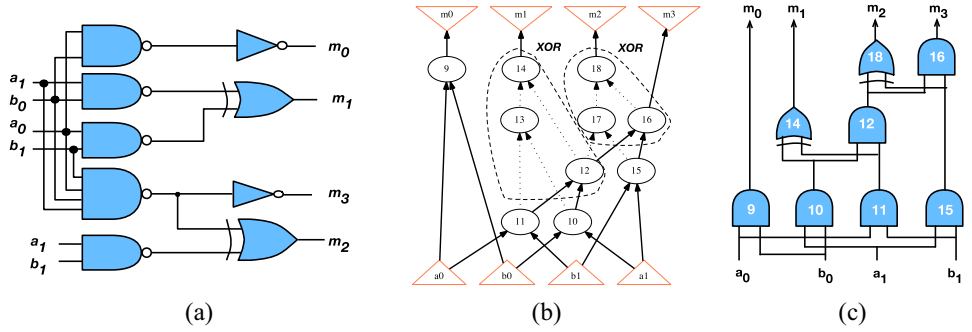

Fig. 2. Two-bit multiplier. (a) Post-synthesized gate-level netlist. (b) AIG of the multiplier. (c) Detected unobserved functions and their relationship to the AIG nodes.

Algorithm 1 Algebraic Rewriting in AIG

Input: Gate-level netlist, output signature Sig_{out}
Output: *Pseudo-Boolean* expression extracted by rewriting
1: Structural hashing the gate-level netlist into AIG, denoted *G(V, E)*.
2: Detect all XOR3 and MAJ3 nodes in *G(V, E)*.
3: Pair the XOR3 and MAJ3 if they have identical signals, denoted as *P*.
4: Topological sort *G(V,E)* considering each element in *P* as one node.
5: $i = 0$; $F_i = Sig_{out}$
6: **while** there are no elements remained in reverse topological order **do**
7: Rewrite: $F_{i+1} = F_i$ by substituting the variables with algebraic equations;
8: $i = i + 1$
9: **return** $F = F_i$ (to be compared with Sig_{in})

when rewriting x_1 and x_2 together, four nonlinear terms are eliminated, namely $2ab$, $2bc$, $2ac$, and $4abc$, generated by the algebraic models of XOR3 and MAJ3. However, if rewriting is applied directly to the gate-level netlist, its efficiency is lost when the MAJ3 and XOR3 functions are mapped into other standard cells by logic synthesis and technology mapping. For example, the XOR3 function mapped using standard cells is shown in Fig. 1(a). In this case, there is no ordering that provides the required polynomial reductions.

III. Approach

This section presents the algebraic rewriting approach based on AIGs. Similarly to [1], the algebraic rewriting process rewrites the output signature for all AIG nodes in a reverse topological order. As discussed in Section II-E, the rewriting order that provides a large number of polynomial reductions, has significant impact on the rewriting performance. However, there are many reverse topological orders available in an AIG, since many nodes can have the same topological depth. This approach automatically detects a reverse topological order for algebraic rewriting that provides maximum polynomial reduction. This is achieved by detecting pairs of MAJ3 and XOR3 nodes using AIG-based *cut enumeration*.

A. Outline of the Approach

The proposed flow is outlined in Algorithm 1. The inputs to the algorithm are: the gate-level netlist and the output signature Sig_{out}. The flow includes three basic steps: 1) converting the gate-level implementation into AIG; 2) detecting all pairs of XOR3 and MAJ3 functions with identical inputs in the AIG; topological sorting the AIG nodes while considering the detected pairs as one element; and 3) applying algebraic rewriting from POs to PIs following the reverse topological order determined in step 2). Note that XOR2 and MAJ2(AND2) are the special cases of XOR3 and MAJ3, where one of the inputs is constant zero. The second step is performed as follows.

1) Computing all three-feasible (three-input) cuts of all AIG nodes.
2) Computing truth tables of all cuts.
3) Storing cuts in the hash table by their ordered input set.
4) Detecting pairs of three-input cuts with identical inputs belonging to different nodes, such that the Boolean functions of the two cuts with the shared inputs belong to the *NPN* classes of XOR3 and MAJ3, respectively.

Note that, in this approach, matching the XOR3 and MAJ3 nodes does not require the inputs and outputs polarity to be the same. Instead, all the cut-points are matched without considering their complemented attributes. For example, instead of being an exact XOR3, the function of a three-feasible cut can be either XOR3 or XNOR3. Similarly, instead of being exactly MAJ3, the function can be one of the eight functions forming an *NPN* class of MAJ3 [19]. To compute the cuts, the three-input cut enumeration is performed in a topological order as described in [20]. The truth tables of the cuts are obtained as a by-product of the cut enumeration. When two fanin cuts merged during the cut computation result in a three-feasible cut, their truth tables are combined according to the logic function of the resulting cut. For the case of three-input cuts, a dedicated precomputation reduces the runtime of truth table computation to a small fraction of that of cut enumeration.

As soon as the XOR3 and MAJ3 pairs are detected, algebraic rewriting is applied to the AIG network in a constrained reverse topological order, in which each XOR3 and MAJ3 pair is considered as one element. This means that at one topological depth, whenever either XOR3 or MAJ3 node of a pair (or its complement) is rewritten, the subsequent rewritten node is of the other type. The AIG nodes with the same topological depth that do not belong to any pair are ordered in the decreasing order of their integer IDs. The algebraic rewriting ends when all elements in AIG network have been rewritten. The algorithm returns the extracted input signature.

Example 1 (2-Bit CSA-Multiplier): The mapped gate-level netlist of a 2-bit CSA-multiplier is shown in Fig. 2(a). First, the gate-level netlist is converted to an AIG representation, Fig. 2(b). Then, a set of XOR3 nodes X, and a set of MAJ3 nodes M are detected: $X = \{14, 18\}$, $M = \{12, 16\}$. Node 14 is XOR3(10, 11, 0) and node 12 is MAJ3(10, 11, 0), where nodes 10 and 11 and constant zero 0 are the inputs; node 18 is XOR3(12, 15, 0) and node 16 is MAJ3(12, 15, 0). Hence, two pairs of XOR3 and MAJ3 are generated, namely, (14, 12) and (18, 16). The order of rewriting is determined as follows: 1) node 18 is the node with highest depth; it is detected as XOR3 and paired with a MAJ3 node 16; hence, the first rewriting starts from nodes 18 and 16, and ends at nodes 12 and 15; 2) similarly to the first rewriting, the second rewriting starts from nodes 14 and 12, and ends at nodes 11 and 10; and 3) the remaining AIG nodes are ordered by their index value in decreasing order. The logic network after detecting all XOR3 and MAJ3 functions are shown in Fig. 2(c).

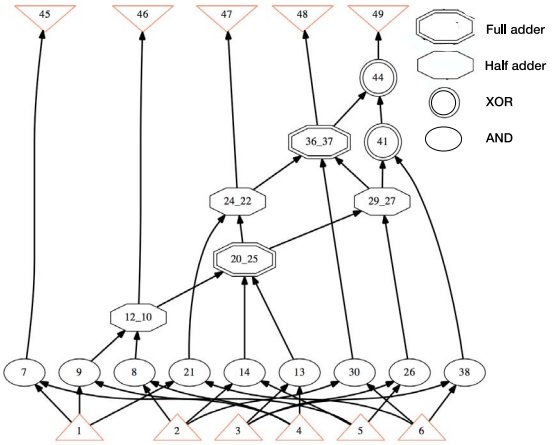

Fig. 3. Detecting MAJ3-XOR3 of a 3-bit post-synthesized CSA-multiplier with MSB z_5 deleted.

B. Detecting Redundant Polynomials

Significant simplification of polynomial construction can be achieved not only by performing algebraic rewriting using a reverse topological order, as discussed above but also by detecting redundant polynomials, such as don't-care polynomials and *vanishing* polynomials [5], [7], [21]. Vanishing polynomials are those that always evaluate to zero; vanishing *monomials* used in the work of [5] are examples of such polynomials. Don't care polynomials can be identified in circuits (such as multipliers) with truncated outputs, where arithmetic operators are truncated to reduce power consumption or delay of the critical path. The removed signals in those circuits contain algebraic information needed to cancel algebraic terms of the remaining output bits. The polynomial associated with the most significant bit (MSB) of an adder or a multiplier is an example of such a polynomial. Note that the logic obtained by removing such output bits is either a carry-out or a sum function of a full adder, implemented by MAJ3 and XOR3 functions. Hence, using the approach of detecting pairs of XOR3 and MAJ3 (Section III-A), the XOR3/MAJ3 nodes that do not belong to any such pairs can also be identified. For example, in a CSA-multiplier with a MSB removed, MAJ3 with same inputs as an unpaired XOR3 is missing. Since one pair of XOR3 and MAJ3 forms a full adder, removing the carry bit (and the MAJ3) makes the function an addition *modulo 2*. In this case, the algebraic model of XOR3 reduces to $a+b+c-2ab-2ac-2bc+4abc \ mod\ 2$ (Table I). The monomials are redundant if their coefficients modulo 2 reduce to 0. In other words, the algebraic model of $a \oplus b \oplus c$ in this case becomes $a+b+c$. Here, the removed terms, $-2ab-2ac-2bc+4abc$, are the required redundant polynomials.

Example 2 (3-Bit CSA-Multiplier With MSB z_5 Deleted): The AIG after detecting XOR3 and MAJ3 pairs of a 3-bit synthesized CSA-multiplier with MSB deleted is shown in Fig. 3. The detected XOR3 and MAJ3 pairs are represented using the ID of the root node of the XOR3 and MAJ3 nodes. We can see that there is one XOR3 (composed of two XOR2 nodes, 41 and 44) with inputs i_{36_37}, i_{27_29}, and i_{38}, that cannot be paired with any MAJ3. This is simply because the synthesis process removed the redundant logic (last carry out) when the MSB has been removed. In this case, the algebraic model of that XOR3 reduces to $z_4(i_{49}) = (i_{36\_37} + i_{27\_29} + i_{38})$.

IV. Results

The technique described in this paper was implemented in the ABC environment [17]. It applies algebraic rewriting to the AIG and generates the polynomial signature. The experiments were conducted on a PC with Intel Xeon CPU E5-2420 v2 2.20 GHz x12 with 32 GB memory. The experiments include gate-level multipliers of different types, up to 512 bits. The results are compared with two leading techniques, *functional extraction* [1] and the *Groebner Basis based* approach with vanishing monomials [5], on synthesized circuits. The results show that the proposed technique is more efficient than these techniques.

TABLE II
EXPERIMENTAL RESULTS COMPARED TO TECHNIQUES OF [1], [5]. *$t(s)$ IS THE RUNTIME IN SECONDS. *mem IS THE MEMORY USAGE IN MB. B = BOOTH MULTIPLIER. WT = WALLACE-TREE. CLA = CARRY LOOK-AHEAD. RC = RIPPLE CARRY. TO = TIME OUT (24 h)

# operand bits	*Pre-Syntheized*				*Post-Synthesized*			
	[1]		This approach		[1]	[5]	This approach	
	t(s)	mem	t(s)	mem	t(s)	t(s)	t(s)	mem
64	1.9	74	0.08	34	5.50	593	0.11	34
128	8.1	288	0.78	117	39.6	TO	0.91	120
128WT-RC	-	-	-	-	MO	TO	0.91	120
128WT-CLA	-	-	-	-	MO	TO	-	MO
256	32.6	1157	7.80	441	285	TO	8.23	439
256B	-	MO	33.7	423	MO	TO	39.5	431
256B-CLA	-	MO	-	MO	-	-	-	MO
512	130	4427	31.7	1695	-	-	-	MO

TABLE III
RESULTS OF APPLYING AIG-BASED ALGEBRAIC REWRITING TO POST-SYNTHESIZED COMPLEX ARITHMETIC CIRCUITS COMPARED TO *Functional Extraction* [1]. MO = MEMORY OUT (8 GB)

Benchmarks (256-bit)	[1]		This approach	
	runtime(s)	mem(MB)	runtime(s)	mem(MB)
$F=A\times B+C$	179.1	1182	5.1	447
$F=A\times(B+C)$	209.3	1120	5.1	451
$F=A\times B\times C$	-	MO	37.5	2871
$F=1+A+A^2+A^3$	-	MO	47.1	3331

Our AIG-based algebraic rewriting method was evaluated using original (presynthesized) and post-synthesized multipliers, shown in Table II; and on post-synthesized arithmetic datapath circuits, shown in Table III. Booth multipliers are generated by ABC [17] using *%blast-b* command. Wallace-tree multipliers are obtained from [5]. The CSA and CLA multipliers are taken from [1]. The CPU runtime reported in the tables includes the entire verification process, i.e., adder-tree extraction and rewriting. The bit-width of the circuits varies between 64 and 512 bits.[2] We can see that the runtime of the proposed approach outperforms *functional extraction* of [1] for the post-synthesized multipliers for any bit-width. The memory usage has been reduced on average 60%, compared to [1]. However, the polynomial signatures of CLA-based multipliers failed to be extracted because of the polynomial explosion. This is because there are many AIG nodes that cannot be identified as pairs of XOR/MAJ functions in CLA-based multipliers. We should emphasize that the run-time complexity of our method for post-synthesized circuits is comparable to that of the original, presynthesized circuits. This is in contrast to [1], which suffered from a "fat belly" effect on heavily synthesized circuits. As an example, extracting functional specification of the post-synthesized 256-bit multiplier requires 9× shorter runtime and less memory than the functional extraction of [1]. Finally, we observed that the rewriting process of this approach takes about 10% of the entire process time, the majority of computation being spent on AIG manipulation and XOR3/Maj3 function extraction.

V. Conclusion

This paper presented a method to improve the efficiency of algebraic rewriting used in arithmetic verification. The method is based on AIG representation of the Boolean network. This approach allows for

[2]512-bit post-synthesized multipliers are not reported in [1].

formal verification of practical multipliers that are heavily optimized and mapped using 14nm technology library. Another contribution of this paper is a technique that automatically detects redundant polynomials to reduce the complexity of algebraic rewriting. In its current version, the technique is applicable to those arithmetic circuits where detection of adder-tree is possible. Verification of CLA-based multipliers, where such a detection is incomplete, is more challenging and is part of the ongoing work.

ACKNOWLEDGMENT

The authors would like to thank Dr. A. Sayed-Ahmed, for providing comparison results, and Prof. A. Biere for providing benchmarks.

REFERENCES

[1] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *Proc. 52nd DAC*, San Francisco, CA, USA, 2015, pp. 52–57.

[2] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 9, pp. 1409–1420, Sep. 2013.

[3] E. Pavlenko *et al.*, "STABLE: A new QF-BV SMT solver for hard verification problems combining Boolean reasoning with computer algebra," in *Proc. DATE*, Grenoble, France, 2011, pp. 155–160.

[4] F. Farahmandi and B. Alizadeh, "Groebner basis based formal verification of large arithmetic circuits using Gaussian elimination and cone-based polynomial extraction," *Microprocess. Microsyst.*, vol. 39, no. 2, pp. 83–96, 2015.

[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 *Proc. DATE*, Dresden, Germany, 2016, pp. 1–6.

[6] C. Yu, W. Brown, D. Liu, A. Rossi, and M. J. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 12, pp. 2131–2142, Dec. 2016.

[7] C. Yu and M. Ciesielski, "Formal verification using don't-care and vanishing polynomials," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Pittsburgh, PA, USA, 2016, pp. 284–289.

[8] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Proc. 43rd DAC*, San Francisco, CA, USA, 2006, pp. 532–535.

[9] R. E. Bryant and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Proc. DAC*, San Francisco, CA, USA, 1995, pp. 535–541.

[10] M. Ciesielski, P. Kalla, and S. Askar, "Taylor expansion diagrams: A canonical representation for verification of data flow designs," *IEEE Trans. Comput.*, vol. 55, no. 9, pp. 1188–1201, Sep. 2006.

[11] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *Proc. DATE*, Munich, Germany, 2001, pp. 114–121.

[12] N. Sorensson and N. Een, "MiniSAT v1. 13—A sat solver with conflict-clause minimization," in *Proc. SAT*, 2005, p. 53.

[13] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013," in *Proc. SAT Competition*, 2013, pp. 51–52.

[14] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *J. Satisfiability Boolean Model. Comput.*, vol. 9, pp. 53–58, 2015.

[15] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. TACAS*, Budapest, Hungary, Mar./Apr. 2008, pp. 337–340.

[16] T. Pruss, P. Kalla, and F. Enescu, "Efficient symbolic computation for word-level abstraction from combinational circuits for verification over finite fields," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 7, pp. 1206–1218, Jul. 2016.

[17] A. Mishchenko *et al.* (2007). *ABC: A System for Sequential Synthesis and Verification*. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc

[18] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," EECS Dept., Univ. California at Berkeley, Berkeley, CA, USA, ERL Tech. Rep., 2005.

[19] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Kyoto, Japan, 2013, pp. 310–313.

[20] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," in *Proc. FPGA*, Monterey, CA, USA, 1998, pp. 35–42.

[21] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *Proc. FMCAD*, Vienna, Austria, 2017, pp. 23–30.