Verification of Gate-level Arithmetic Circuits by Function Extraction

M. Ciesielski, C. Yu, D. Liu, and W. Brown
University of Massachusetts, Amherst, USA
{*ciesiel, ycunxi, duo, webrown*}*@umass.edu*

André Rossi*
*Université de Bretagne-Sud, Lab STICC, France
andre.rossi@univ-ubs.fr

Abstract - The paper presents an algebraic approach to functional verification of gate-level, integer arithmetic circuits. It is based on extracting a unique bit-level polynomial function computed by the circuit directly from its gate-level implementation. The method can be used to verify the arithmetic function computed by the circuit against its known specification, or to extract the arithmetic function implemented by the circuit. Experiments were performed on arithmetic circuits synthesized and mapped onto standard cells using ABC system. The results demonstrate scalability of the method to large arithmetic circuits, such as multipliers, multiply-accumulate, and other elements of arithmetic datapaths with up to 512-bit operands and over 2 Million gates. The procedure has linear runtime and memory complexity, measured in the number of logic gates of the design.

I. Introduction

Despite a considerable progress in verification of random and control logic, advances in formal verification of arithmetic designs have been lagging. This can be contributed mostly to the difficulty in efficient modeling of arithmetic circuits and datapaths without resorting to computationally expensive Boolean methods that require "bit blasting", i.e., flattening the design to a bit-level netlist.

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. While some EDA vendors offer tools that generate "correct by construction" arithmetic components, the problem of verifying non-standard, bit-optimized embedded arithmetic circuits remains open. Furthermore, while it could be argued that industry somehow manages to handle the arithmetic verification problems, it does it at a high cost by throwing at it all possible verification methods and tools, both formal, and simulation-based [1]. This sentiment is also echoed in the most recent paper on floating point verification at IBM [2].

The work presented in this paper aims at overcoming some of these problems. It addresses the verification problem at an algebraic level, treating an arithmetic circuit and its specification (if known) as a properly constructed algebraic system. The proposed technique solves the verification problem by *function extraction*, i.e., by deriving arithmetic function computed by the circuit from its low-level circuit implementation. The method can be used to verify the extracted function against the given specification (if known), or as a reverse engineering tool, to learn the function performed by the circuit. In case of an incorrectly implemented function, this method will generate a counterexample (bug trace).

II. Related Work

Several approaches have been proposed to check an arithmetic circuit against its functional specification. Different variants of canonical, graph-based representations have been proposed, including Binary Decision Diagrams (BDDs), Binary Moment Diagrams (BMDs) [3], Taylor Expansion Diagrams (TED) [4], 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 requirement for complex arithmetic circuits, such as multipliers.

Arithmetic verification problems have been typically modeled using Boolean satisfiability (SAT) or satisfiability modulo theories (SMT). Several SAT solvers have been developed to solve Boolean decision problems, including ABC, MiniSAT, and others. Some of them, such as CryptoMiniSAT [5], specifically target XOR-rich circuits, but, like all others, are based on a computationally expensive DPLL decision procedure. Several techniques combine linear arithmetic constraints with Boolean SAT in a unified algebraic domain [6]; or combine automatic test pattern generation (ATPG) and modular arithmetic constraint-solving techniques for the purpose of test generation and assertion checking [7]; but they do not offer sufficient scalability. Approaches based on ILP models of the arithmetic operators [8] [9] are also known to be computationally expensive and not scalable.

SMT solvers depart from treating the problem in a strictly Boolean domain and integrate different well-defined theories (Boolean logic, bit vectors, integer arithmetic, etc.) into a DPLL-style SAT decision procedure [10]. Some of the most effective SMT solvers potentially applicable to our problem are Boolector, Z3, CVC and Yices, among others. However, SMT solvers still model the problem as a decision problem and are not efficient at solving verification problems that appear in arithmetic circuits.

Another class of solvers include Theorem Provers, deductive systems for proving that an implementation satisfies the specification, using mathematical reasoning. The proof system is based on a large and strongly problem-specific database of axioms and inference rules, such as simplification, rewriting, induction, etc. Some of the most popular theorem proving systems are: HOL, PVS, and Boyer-Moore/ACL2, Nqthm.The success of verification using theorem prover depends on the set of available axioms and rewrite rules, and on the choice and order in which the rules are applied during the proof process, with no guarantee for a conclusive answer. Similarly, term rewriting techniques, such as [11] or [12], are incomplete and "may fail to generate the proof because additional lemmas are needed" [12].

One of the most advanced techniques that have potential to solve the arithmetic verification problem are those based on symbolic Computer Algebra [13]. These methods model the arithmetic circuit specification and its hardware implementation as polynomials [14],[15],[16],[17],[18],[19]. They attempt to prove that the implementation satisfies the specification by performing a series of divisions of the specification polynomial F by the implementation polynomials $B = \{f_1, \ldots, f_s\}$, representing the circuit components. For example, the specification of an integer multiplier circuit $Z = X \cdot Y$ is $F = Z - X \cdot Y$. The implementation polynomials for gate-level circuits are derived from logic gate equations, similar to those shown in eq.(1).

The verification problem employed by these methods is posed as the reduction of F modulo B, denoted $F \xrightarrow{B}_{+} r$. If $r = 0$ the implementation satisfies the specification. However, if $r \neq 0$, such a conclusion cannot be made: B may not be sufficient to reduce F to 0, and yet the circuit may be correct. To check if F is reducible to zero, one must use a *canonical* set of polynomials, $G = \{g_1, ..., g_t\}$, called *Groebner basis*, obtained from a linear combination of polynomials $f_i \in B$ with polynomial coefficients. In general, a linear combination of polynomials f_i is called an *ideal* $J = \langle f_1, ..., f_s \rangle$. The resulting reduction problem is referred to as *ideal membership testing*: check $F \in J$. [1]

Wienand et. al. [16] model an arithmetic circuit as an *arithmetic bit-level* (ABL) network of adders and other arithmetic operators. Both the specification and the operators are represented as polynomials over $\mathbb{Z}_{2^n}$. They show that, the properly ordered set G of polynomials representing logic gates automatically renders it a Groebner basis. The verification problem is solved by testing if specification F reduced modulo G vanishes over $\mathbb{Z}_{2^n}$ using a computer algebra system, Singular [20]. In [17], the solution is further restricted to variables in $\mathbb{Z}_2$ and the reduction formulated directly over quotient ring $Q = \mathbb{Z}_{2^n}[X]/\langle x^2 - x \rangle$. Here, the ideal $\langle x^2 - x \rangle$ is the constraint restricting variables x to Boolean. While mathematically elegant, adding this constraint for all variables makes the method computationally expensive for gate-level circuits. In general, the method of [17] is limited to ABL networks composed of half adders HA.

Lv, Kalla, et. al [18], [19], formulated the verification problem similarly, but applied it to Galois field (GF) arithmetic circuits, which enjoy certain simplifying properties. Specifically, for GF, the problem reduces to the ideal membership testing over a larger ideal that includes $J_0 = \langle x^2 - x \rangle$ in $\mathbb{F}_2$. The solution uses a modified Gaussian elimination technique. In [19], a symbolic computer algebra method is used to derive a word level abstraction for GF circuits, where GF operators are elements of a polynomial ring with coefficients in $\mathbb{F}_{2^k}$. This work relies on the customized computation of Groebner basis and applies only to GF networks. It does not extend to polynomial rings in integers $\mathbb{Z}_{2^n}$ which is the subject of this paper.

A different approach to arithmetic verification has been proposed in [21] and [22], where a bit-level network is described by a system of linear equations. The system is then reduced to a single *algebraic signature*, F_{Sig}, using standard linear algebra methods and compared to the specification polynomial F_{spec}. A non-zero *residual expression* $RE = F_{Sig} - F_{spec}$, determines a potential mismatch between the implementation and the specification, indicating a design error. Additional step is needed to check if $RE = 0$, which may be as difficult as the original problem itself. An extension to this work has been recently presented in [23], by computing input signature from the known output signature using a *network-flow approach.* This technique also relies on the half-adder based circuit structure and represents logic gates as elements of HAs. Logic gates that cannot be mapped into adders are represented as HAs, with an unused output left as "floating". Additional constraint relating floating signals to fanouts in the circuit must be satisfied for the result to be trusted; however the computation to verify this condition can be expensive. For this reason, this method becomes inefficient if the number of logic gates dominates the HA network. Also, the circuit would need to be partitioned into linear and non-linear portions, which is a non-trivial task.

In summary, the problem of formally verifying integer arithmetic circuits over integers $\mathbb{Z}_{2^n}$ remains open [24]. To the best of our knowledge, the techniques reviewed here cannot efficiently solve the verification problem for gate-level arithmetic circuits in $\mathbb{Z}_{2^n}$ over Boolean variables $\mathbb{Z}_2$, which is the problem we describe in this paper.

III. Function Extraction

Novelty and Contribution: This paper offers a robust solution to arithmetic verification by extracting a *unique* bit-level polynomial function implemented by the circuit, directly from its gate-level implementation. This is done by transforming the polynomial representing encoding of the outputs (called *output signature*) into a polynomial at the primary inputs (the *input signature*). If the specification of the circuit is known, the extracted input signature will be compared with that specification. Otherwise, the computed signature provides the arithmetic function implemented by the circuit.

The method uses an efficient algebraic model of the circuit, with logic gates represented by algebraic expressions, while correctly modeling signals as Boolean variables. In contrast to [23], it works directly on unstructured, gate-level implementations. And in contrast to [17],[19] and other computer algebra methods, it is done using efficient polynomial transformation process, without a need for expensive Groebner-based polynomial division.

To the best of our knowledge, this approach has not been attempted before in the context of gate-level integer arithmetic in $\mathbb{Z}_{2^n}$ [2]. In a sense, it indirectly proves that checking if the implementation satisfies the specification can be done using ideal membership testing in $\mathbb{Z}_{2^n}$, although such a result is not known in theory [24].

[1] In general, one must test if $F \in I(V(J))$. It is only for finite fields $\mathbb{F}_{2^q}$ that this test reduces to $F \in J$. Details can be found in [13] [18].

[2] The functional abstraction technique described in [19] applies only to Galois field circuits and is based on polynomial reduction via Groebner basis.

A. Algebraic Model

The circuit is modeled as a network of logic elements of arbitrary complexity: basic logic gates (AND, OR, XOR, INV) and complex (AOI, OAI, etc.) standard cells obtained by synthesis and technology mapping.

Each logic element is modeled as a *pseudo-Boolean polynomial* f_i, with variables from $\mathbb{Z}_2$ (binary) and coefficients from $\mathbb{Z}_{2^n}$ (integers modulo 2^n). The following algebraic equations are used to describe Boolean operators and arbitrary logic gates:

$$\begin{aligned} \neg a &= 1 - a \\ a \wedge b &= a \cdot b \\ a \vee b &= a + b - a \cdot b \\ a \oplus b &= a + b - 2a \cdot b \end{aligned} \tag{1}$$

In our model, the arithmetic function computed by the circuit is specified by two polynomials: an input signature and an output signature. The *input signature*, Sig_{in}, is a polynomial in primary input variables that uniquely represents the integer function computed by the circuit, i.e., its *specification*. For example, an n-bit binary adder with inputs $\{a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1}\}$, is described by $Sig_{in} = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i$. Similarly, the input signature of a 2-bit signed multiplier, shown in Fig. 1, is $Sig_{in} = (-2a_1 + a_0)(-2b_1 + b_0) = 4a_1b_1 - 2a_0b_1 - 2a_1b_0 + a_0b_0$, etc. In our approach, the input specification need not to be known; it will be derived from the circuit implementation as part of the verification process.

Similarly, the *output signature*, Sig_{out}, of the circuit is defined as a polynomial in the primary output signals. Such a polynomial is uniquely determined by the n-bit encoding of the output, provided by the designer. For example, the output signature of the 2-bit signed multiplier in Fig. 1 is $-8z_3 + 4z_2 + 2z_1 + z_0$. In general, an output signature of an unsigned arithmetic circuit with n output bits z_i is represented as a linear polynomial, $Sig_{out} = \sum_{i=0}^{n-1} 2^i\ z_i$. Similar expression is derived for signed arithmetic circuits.

Our goal is to transform the output signature, Sig_{out}, using polynomial representation of the internal logic elements, into the input signature, Sig_{in}. By construction, the resulting Sig_{in} will contain only the primary inputs (PI) and will uniquely determine the arithmetic function computed by the circuit (*c.f.* Theorem 1).

B. Outline of the Approach

The proposed approach is illustrated with a simple 2-bit signed multiplier example, shown in Fig. 1. The input signature Sig_{in} is computed by by a series of transformations performed in a reversed topological order, from the primary outputs (PO) to primary inputs (PI), starting at the known output signature $F_0 = Sig_{out}$. Each equation corresponds to a *cut* in the circuit, i.e., a set of signals that separate primary inputs from primary outputs. The following sequence is given here for the purpose of illustration, with the discussion on the ordering of transformations to follow. First, F_0 is transformed into F_1 using substitutions $z_3 = 1 - x_8$ and $z_2 = 1 - x_9$. Subsequently, F_2 is obtained from F_1 using equations for x_8 and x_9, and so on, culminating at the primary inputs with $F_7 = 4a_1b_1 - 2a_0b_1 - 2a_1b_0 + a_0b_0$.

$$\begin{aligned} F_0 &= -8z_3 + 4z_2 + 2z_1 + z_0 \\ F_1 &= 8x_8 - 4x_9 + 2z_1 + z_0 - 4 \\ F_2 &= 8(x_1 + x_7 - x_1x_7) - 4(x_1 + x_7 - 2x_1x_7) + 2z_1 + z_0 - 4 \\ F_3 &= 4x_1 + 4x_7 + 2z_1 + z_0 - 4 \\ F_4 &= 4x_1 + 4x_5x_6 + 2(x_5 + x_6 - 2x_5x_6) + z_0 - 4 \\ F_5 &= 4x_1 + 2(x_5 + x_6) + z_0 - 4 \\ F_6 &= 4x_1 - 2x_2 - 2x_3 + x_4 \\ F_7 &= 4a_1b_1 - 2a_0b_1 - 2a_1b_0 + a_0b_0 \\ &= (-2a_1 + a_0)(-2b_1 + b_0) \end{aligned}$$

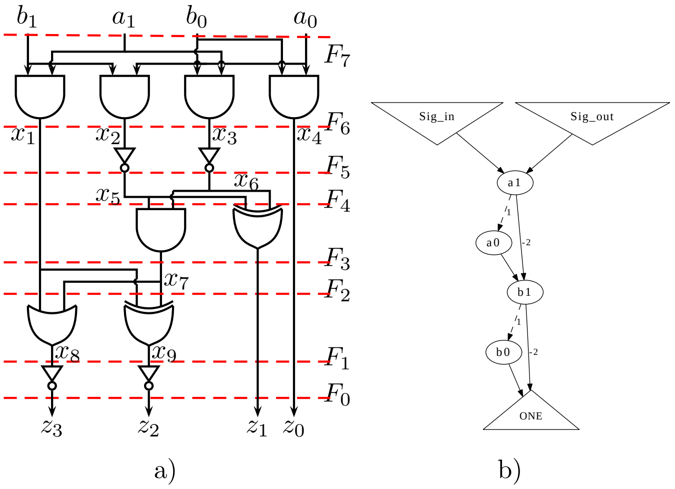

Fig. 1. Verifying a 2-bit signed multiplier: a) Gate-level circuit with output signature $Sig_{out} = -8z_3 + 4z_2 + 2z_1 + z_0$; b) Arithmetic function extracted from the circuit using TED in normal factored form: $Sig_{in} = (-2a_1 + a_0)(-2b_1 + b_0)$.

Note the local increase in the polynomial size (at F_2 or F_4) known as "fat belly" effect, before it gets eventually reduced to the expression in PIs only. The choice of the cuts and the order in which the variables are eliminated by substitution has a profound influence on the size of the fat belly and the efficiency of the method. They are guided by the following heuristics to keep the size of the intermediate expressions as small as possible.

- **Fanouts:** Identify variables that depend on common (fanout) inputs and perform their substitution simultaneously. This increases a chance for eliminating common subexpressions. For example, in Fig. 1 variables in subexpression $8x_8 - 4x_9$ depend on common fanout variables x_1 and x_7. As a result, $8x_8 - 4x_9 = 4(2x_8 - x_9)$ reduces to $4(x_1 + x_7)$, without introducing a nonlinear term $8x_1x_7$. Such nonlinear terms are particularly harmful if their variables continue to be substituted by other variables, potentially leading to exponential explosion.
- **Dependency and Levelization:**

a) Substitution must follow the reverse-topological order; once a given variable (output of a gate) is substituted by an algebraic expression of the gate inputs, it will be eliminated from the current cut expression and will never be considered again. That is, a variable is substituted for only after substituting all signals in its logical cone. For example, before substituting for x_6, one must substitute for x_7 and z_1, since they both depend on x_6. Since the circuit is acyclic, there always exists an ordering of substitutions that satisfies this condition. We refer to this topological constraint informally as "vertical", since it orders variables upwards from POs to PIs.

b) To further increase the efficiency of substitution, another ("horizontal") constraint is imposed on the ordering of the candidate variables at a given transformation step. Specifically, the variables that are at the same logic level (from PIs) and have transitive fan-in to common variables should be eliminated together, as this will maximize a chance of the reduction of common terms. It is these variables that define the best cut at each step of the procedure.

- **Vanishing Polynomials:** In some arithmetic circuits a particular output bit may always evaluate to zero. This is typically associated with MSB, but this is not the only case. For example, in the squarer circuit (multiplier which computes a square value Z of some integer input) the output bit z_1 is always 0. For this reason one may want to exclude bit z_1 from the output signature, $Sig_{out} = \sum_{i=0}^{n-1} 2^i\ z_i$. However, the set of algebraic expressions associated with the term $2z_1$ offers some early simplification during the computation of the signature, before it reaches the primary inputs. Obviously, the logic cone of z_1 itself will reduce to 0 at the PI, but the terms of its intermediate cuts (at internal signals) help reduce the size of the intermediate cuts of the rest of the circuit. We refer to such a redundant expression as the *vanishing polynomial*, as it vanishes (evaluates to 0) for all possible values of its input variables. We comment in Section V on how presence of such a redundancy may help the verification.
- **Complex gates:** Our signature transformation algorithm works on a fabric of basic Boolean gates; this offers high logic granularity and the greatest choice of signals for the selection of the smallest cut. For the design with complex gates (standard cells AOIxx, OAIxx, etc.), algebraic equations are written for each internal signal of the gate, rather than only for its output. As confirmed by our experiments, this offers a richer set of cuts to choose from and increases a chance of an earlier simplification of the cut expression.
- **Binary signals:** During elimination, the expensive division by the ideal $\langle x^2 - x \rangle$, employed by [17], is replaced by lowering x^k to x every time variable x is raised to higher degree during the substitution process. For example, if at any point an expression contains a term xyx, it will be replaced by xy. With this, an expression, such as $xyx - yxy$, will immediately reduce to 0.

Efficient Datastructure: Our algorithm uses an efficient data structure to support these simplifications and efficiently implement an iterative substitution and elimination process. Specifically: a data structure is maintained that records the terms in the expression that contain the variable to be substituted. It reduces the cost of finding what terms will have their coefficients changed during the substitution. The expression data structure is a C++ object that represents a pseudo-Boolean expression. It supports both fast addition and fast substitution with two C++ maps, implemented as binary search trees, a *terms map* and a *substitution map.*

C. Properties of the Extracted Input Signature

Once Sig_{in} has been computed, it is analyzed to see if it matches the expected specification. The comparison between the two expressions can be done using canonical data structures, such as BMD [3] or TED [4] that can check equivalence between two word-level outputs expressed in bit-level inputs.

In case of a buggy circuit, if the specification is given and the system can successfully compute input signature, then any mismatch between the specification and input signature can be used to generate a counter-example (bug trace). This can be done, by solving a SAT/SMT problem on that mismatch polynomial. Any satisfying solution will provide a test vector for the counter-example.

When the specifiction is not given, TED can represent the function implemented by the circuit in normal factored form to help identify the type of arithmetic function obtained. TED has a capability of finding best ordering of variables from which such a form can be obtained [25]. In large arithmetic circuits, additional variable ordering directives may be given by the designer if the bit-level composition of input words is known. For example, for the circuit in Fig. 1(a), the input signature computed by our method is $Sig_{in} = 4a_1b_1 - 2a_0b_1 - 2a_1b_0 + a_0b_0$. Its TED representation shown in Fig. 1(b) reveals the canonical factored form, $Sig_{in} = (-2a_1 + a_0)(-2b_1 + b_0)$. This indicates that the function computed by the circuit is a two-bit signed multiplier, $A \cdot B$, if the variables (a_1, a_0) and (b_1, b_0) form the two-bit input words, A and B.

Essential part of the described approach is the following theoretical result about the correctness and uniqueness of the computed input signature. Here, "correct" means that the result is the same as if it were computed with Boolean methods. This result applies to *combinational circuits*, but it can be readily extended to sequential circuits by unrolling the circuit over a fixed number of time frames into a combinational circuit (bounded model).

Theorem: *Given a combinational circuit composed of basic logic gates, described by polynomial expressions (1), input signature Sig_{in} computed by the proposed procedure is unique and correctly represents the arithmetic function implemented by the circuit.*

Proof: The proof of correctness hinges on the fact that each internal signal is correctly represented by an algebraic expression, i.e., such an expression evaluates to a *correct Boolean value*. Specifically, it can be easily verified that equations (1) are the correct algebraic representations of basic Boolean functions. Hence, any logic function that is expressed recursively by eq. (1) must evaluate to a correct Boolean value; once the polynomial is reduced by removing redundant terms, the algebraic representation is unique. Example: XOR function, $f = a \oplus b = a'b + ab'$, can be written as $f = (1-a)b + a(1-b) - ((1-a)b)(a(1-b))$, which reduces to a unique form, $a + b - 2ab$. Hence, a PO signal is correctly represented by variables in its logic cone, up to the primary inputs. Therefore, Sig_{out}, which is the weighted sum of the output signals, is eventually replaced by Sig_{in}. For this reason such computed Sig_{in} is a correct algebraic representation of the circuit.

The proof of uniqueness is based on induction on i, the step when polynomial F_i is transformed into F_{i+1}.

Base phase: polynomial $F_0 = Sig_{out}$, a linear combination of primary outputs, is unique. Also, as discussed above, algebraic representation of each logic gate is unique.

Induction phase: Assuming that F_i is unique, we must prove that F_{i+1} is unique. Recall that each variable in F_i represents output of some logic gate; during the transformation process it is substituted by a unique polynomial of that gate. Since the circuit is combinational (no loops) and the substitution is done in reversed topological order, at each step i a variable in F_i is replaced by a unique polynomial in new variables. Hence, polynomial F_{i+1} derived from F_i by such substitution is also unique. □

IV. Experimental Results

The verification technique described in this paper was implemented in C++. The program was tested on a number of gate-level combinational arithmetic circuits, taken from [26]: CSA multipliers, add-multiply, matrix multipliers, squaring, etc., with operands ranging from 64 to 512 bits. The results are shown in Table I and II. The experiments were conducted on a PC with Intel Processor Core i5-3470 CPU 3.20GHz x4 with 15.6 GB memory.

Experimental setup: The gate-level structures were obtained by direct translation of standard implementation of the designs onto basic logic gates [26]. The designs labeled with extension *.syn* were synthesized and mapped using ABC system [27] (commands: *strash; logic; map*) onto *mcnc.genlib* standard cell library. ABC was unable to synthesize the 512-bit CSA multiplier due to memory limitation.

SAT experiments: the miter for the multiplier design was created using ABC, command *miter*, with the reference design generated by ABC with *gen -N -m* command. ABC was also tested using the combinational equivalence checking command *cec*. SMT experiments: given the specification Sig_{in} and output encoding Sig_{out}, the goal was to prove that $(Sig_{out} - Sig_{in})$ is unsatisfiable (unSAT). Two types of encoding of the gate equations were tested: 1) Algebraic gate equations written in SMT2 format; and 2) The CNF formula for the gates, produced by ABC, translated using online parser into SMT2. This encoding had much better performance than method (1) for all the tested SMT solvers. These results are shown in Table II.

The plot for CPU runtime in Fig. 2 shows an approximately *linear* runtime complexity of the program in the number of gates for all the tested circuits. This should be contrasted with quadratic runtime complexity of [23] (col. 5) and the exponential time complexity of other tools.

Table II gives comparison of our results for the synthesized multipliers with winners of recent SMT competitions and evaluation, including Boolector, Z3, CVC4; best SAT tools and the ABC system; with the symbolic algebra tool, SINGULAR; and Synopsys' *Formality* system. It shows that our technique surpasses those tools in CPU time by several orders of magnitude.

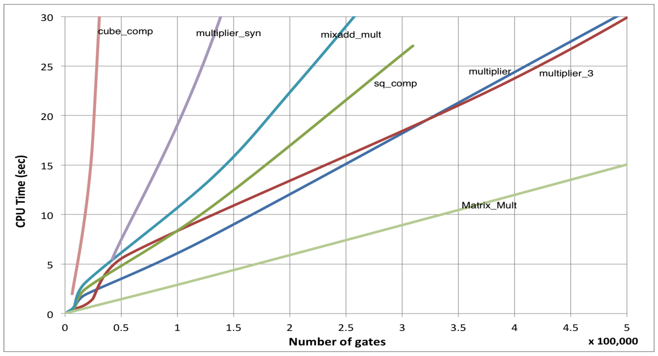

Fig. 2. CPU time for verifying arithmetic combinational circuits.

V. Conclusions and Future Work

The paper describes an efficient approach to derive the function computed by an integer arithmetic circuit from its gate-level implementation. It indirectly shows that the test if the implementation satisfies the specification can, in fact, be done using ideal membership testing in $\mathbb{Z}_{2^n}$; but in contrast to methods based on computer algebra it is implemented as an efficient signature transformation process. This by itself is an important theoretical contribution to the field [24]. Also note that the SAT and SMT approaches can only be used for verification, and not for arithmetic function extraction provided by our method.

Our approach uses an advanced data structure and a set of efficient heuristics to effect this extraction. The results show that the approach can handle gate-level integer multiplier circuits up to 512 bits and containing over 2 million gates. It should be noted, that our experiments involved circuits synthesized with ABC onto a relatively simple set of complex gates (*mcnc.genlib*). It seems that the synthesis tool which retains certain degree of redundancy in the circuit may help in the verification (recall the discussion of vanishing polynomials and their effect on "fat belly" in Section III-B). Solving the verification problem for highly optimized bit-level circuits, synthesized with commercial tools, remains a challenge, as these tools are more aggressive in removing the useful redundancy (vanishing polynomials). Future work will concentrate on verifying circuits synthesized with commercial tools; on verifying sequential and floating point circuits; and on debugging.

Acknowledgments. The authors wish to thank Profs David Cox and Priyank Kalla for their invaluable comments about the mathematical models used in the paper. This work has been supported by a grant the National Science Foundation under award CCF-1319496.

References

[1] R. Kaivola, et al. "Replacing testing with formal verification in Intel Core i7 processor execution engine validation," in *Computer Aided Verification*. Springer, 2009, pp. 414–429.

[2] U. Krautz, V. Paruthi, A. Arunagiriand S. Kumar, S. Pujar, and

Benchmark		64-Bit			128-Bit			256-Bit		
Name	Function	# Gates	CPU [sec]	Mem	# Gates	CPU [sec]	Mem	# Gates	CPU [sec]	Mem
adder	$F = A + B$ (Wallace)	445	0.01	1.5 MB	893	0.05	3.5 MB	1.8K	0.10	5.7 MB
adder_syn	$F = A + B$ (Wallace)	559	0.06	3.5 MB	1.1K	0.15	5.3 MB	1.3K	0.22	5.5 MB
shift_Add	$F = A + A/2 + A/4 + A/8$	1.9K	0.09	3.6 MB	3.8K	0.20	9.8 MB	7.7K	0.44	18.2 MB
shift_Add_syn	$F = A + A/2 + A/4 + A/8$	1.2K	0.08	4.6 MB	2.5K	0.28	9.1 MB	4.6K	1.39	15.4 MB
multiplier	$F = A \cdot B$ (CSA Array)	32K	1.89	72 MB	129K	7.78	129 MB	521K	32.26	1.15 GB
multiplier_syn	$F = A \cdot B$ (CSA Array)	42K	5.50	76 MB	164K	39.64	299 MB	663K	285.22	1.25 GB
mixAddMult	$F = A \cdot (B + C)$	33K	3.17	18 MB	131K	13.77	306 MB	525K	70.18	1.18 GB
mixAddMult_syn	$F = A \cdot (B + C)$	39K	5.03	80 MB	161K	34.32	302 MB	650K	209.31	1.12 GB
multiplier_3	F=$A_1B_1 + A_2B_2 + A_3B_3$	98K	5.88	75 MB	393K	23.32	392 MB	1,571K	-	MO
sq_comp	F=$A^2 + 2 \cdot A + 1$	33K	2.56	18 MB	132K	10.96	285 MB	527K	48.84	1.13 GB
cube_comp	$F = 1 + A + A^2 + A^3$	99K	192.8	416 MB	395K	2052.85	2.3 GB	1,576K	TO	-
Matrix_Mult	F=$A[3 \times 3] \cdot B[3 \times 1]$	293K	18.82	621 MB	1,176K	77.09	2.5 GB	4,712K	-	MO

TABLE I

CPU Time and Memory Results (TO = Timeout after 3600 sec; MO = Memory out of 8 GB)

multiplier_synthesized												*multiplier_3*	
Statistics		Function-Extraction		[23]	SAT [sec]			SMT [sec]			Commercial		
Size	#Gates	CPU [sec]	Mem	[sec]	lingeling	minisat_blbd	ABC	Boolector	Z3	CVC4	Formality	Our	Formality
4	86	0.01	2.2 MB	0.45	0.00	0.00	0.01	0.00	0.03	0.09	0.81	0.02	2.34
8	481	0.04	2.9 MB	1.72	4.40	62.75	11.66	7.18	16.55	42.63	3.19	0.07	21.51
12	1.2K	0.08	4.3 MB	5.21	TO	1615.47	UD	2030.19	TO	TO	108.1	0.17	150.65
16	2.1K	0.14	6.1 MB	7.34	TO	TO	UD	TO	TO	TO	111.2	0.33	798.24
64	41.4K	5.50	76 MB	TO	TO	TO	UD	TO	TO	TO	675.4	5.88	TO
128	164K	39.64	299 MB	TO	TO	TO	UD	TO	TO	TO	TO	23.32	TO
256	663K	285.22	1.25 GB	TO	TO	TO	UD	TO	TO	TO	TO	97.60	TO
512^*	2,091K	130.22	4.44 GB	TO	TO	TO	UD	TO	TO	TO	TO	MO	TO

TABLE II

Results for synthesized multiplier; comparison with [23], SAT, SMT and commercial tools (TO = 3600 sec; MO = 8 GB)

T. Babinsky, "Automatic Verification of Floating Point Units," in *Proc. Design Automation Conference*. ACM, 2014, pp. 1–6.

[3] R. E. Bryant and Y. A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," in *Proc. Intl. Design Automation Conference*, 1995, pp. 535–541.

[4] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *IEEE Trans. on Computers*, vol. 55, no. 9, pp. 1188–1201, Sept. 2006.

[5] M. Soos, "Enhanced Gaussian Elimination in DPLL-based SAT Solvers," in *Pragmatics of SAT*, Edinborough, July 2010.

[6] F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL Models using Linear Programming and 3-Satisfiability," in *Proc. Design Automation Conference*, 1998, pp. 528–533.

[7] C.-Y. Huang and K.-T. Cheng, "Using Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques for Assertion Property Checking," *IEEE Trans. on CAD*, vol. 20, no. 3, pp. 381–391, March 2001.

[8] R. Brinkmann and R. Drechsler, "RTL-Datapath Verification using Integer Linear Programming," in *Proc. Asia and South Pacific Design Automation Conference*, 2002, pp. 741–746.

[9] Z. Zeng, K. Talupuru, and M. Ciesielski, "Functional Test Generation based on Word-level SAT," in *Journal of Systems Architecture*. Aug. 2005, vol. 5, pp. 488–511, Elsevier Publishers.

[10] A. Biere, M. Heule, H. V. Maaren, and T. Walsch, *Satisfiability Modulo Theories in Handbook of Satisfiability*, IOS Press, 2008.

[11] S. Vasudevan, V. Viswanath, R. W. Sumners, and J. A. Abraham, "Automatic Verification of Arithmetic Circuits in RTL using Stepwise Refinement of Term Rewriting Systems," *IEEE Trans. on Computers*, vol. 56, no. 10, pp. 1401–1414, 2007.

[12] D. Kapur and M. Subramaniam, "Mechanical Verification of Adder Circuits using Rewrite Rule Laboratory," *Formal Methods in System Design*, vol. 13, no. 2, pp. 127–158, 1998.

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

[14] T. Raudvere, A. K. Singh, I. Sander, and A. Jantsch, "System Level Verification of Digital Signal Processing application based on the Polynomial Abstraction Technique," in *Proc. Intl. Conf. on Computer-Aided Design*, 2005, pp. 285–290.

[15] N. Shekhar, P. Kalla, and F. Enescu, "Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing," *IEEE Trans. on Computer-Aided Design*, vol. 26, no. 7, pp. 1320–1330, July 2007.

[16] 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.

[17] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G.M. Greuel, "Stable: A new qf-bv smt solver for hard verification problems combining boolean reasoning with computer algebra," in *DATE*, 2011, pp. 155–160.

[18] J. Lv, P. Kalla, and F. Enescu, "Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmatic Circuits," *IEEE Trans. on CAD*, pp. 1409–1420, Sept. 2013.

[19] T. Pruss, P. Kalla, and F. Enescu, "Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases," in *Proc. Design Automation Conference*, 2014, pp. 1–6.

[20] 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.

[21] M. A. Basith, T. Ahmad, A. Rossi, and M. Ciesielski, "Algebraic Approach to Arithmetic Design Verification," in *Formal Methods in CAD*. 2011, pp. 67–71, FMCAD.

[22] M. Ciesielski and A. Rossi W. Brown, "Arithmetic Bit-level Verification using Network Flow Model," in *Haifa Verification Conference, HVC'13*. Nov. 2013, pp. 327–343, Springer.

[23] M. Ciesielski, W. Brown, D. Liu, and A. Rossi, "Function Extraction from Arithmetic Bit-level Circuits," in *IEEE Annual Symposium on VLSI*, July 2014, pp. 356–361.

[24] David Cox, "Private Communication," February 2014.

[25] M. Ciesielski, D. Gomez-Prado, Q. Ren, J. Guillot, and E. Boutillon, "Optimization of Data-Flow computation using Canonical TED Representation," *IEEE Trans. on Computers*, vol. 28, no. 9, pp. 1321–1333, September 2009.

[26] I. Koren, *Computer Arithmetic Algorithms*, University Press, 2002.

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