Formal Verification of Arithmetic Circuits by Function Extraction

Cunxi Yu, *Student Member, IEEE,*, Walter Brown, Duo Liu,
André Rossi, Maciej Ciesielski *Senior Member, IEEE*

***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 an 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 results show that our approach wins over the state-of-the-art SAT/SMT solvers by several orders of magnitude of CPU time. The procedure has linear runtime and memory complexity, measured by the number of logic gates.**

***Index Terms*—Formal verification, functional verification, arithmetic circuits, computer algebraic**

I. Introduction

With an almost unmanageable increase in the size and complexity of ICs and SoCs, hardware design verification have become a dominating factor of the overall design flow. Despite a considerable progress in verification of random and control logic, advances in formal verification of arithmetic designs have been lagging. This can be attributed mostly to the difficulty in efficient modeling of arithmetic circuits and datapaths without resorting to computationally expensive Boolean methods, such as BDDs and SAT, that require "bit blasting", i.e., flattening the design to a bit-level netlist. Approaches that rely on computer algebra and Satisfiability Modulo Theories (SMT) methods are either too abstract to handle the bit-level nature of arithmetic designs or require solving computationally expensive decision or satisfiability problems. Similarly, theorem provers require a significant human interaction and intimate knowledge of the design to guide the proof process.

Importance of arithmetic verification problem grows with an increased use of arithmetic modules in embedded systems to perform computation intensive tasks for multi-media, signal processing, and cryptography applications. EDA vendors, such as Synopsys, do offer tools that automatically generate "correct by construction" arithmetic components that can be used with high level of confidence in commercial designs. However, this practice, acceptable for standard IP components, does not solve the problem of verifying non-standard, highly bit-optimized embedded arithmetic circuits.

The work presented in this paper aims at overcoming some of these limitations. 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). The results presented here are based on an in-depth analysis of recent advances in computer algebra, reviewed in Section II.

II. Related Work

A. Canonical Diagrams

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) [1], Binary Moment Diagrams (BMDs) [2] [3], Taylor Expansion Diagrams (TED) [4], and other hybrid diagrams [5].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. BDDs are being used, along with many other methods, for local reasoning, but not as monolithic data structure [6]. BMDs and TEDs offer a better space complexity but require word-level information of the design, which is often not available or is hard to extract from bit-level netlists.

B. SAT and SMT solvers

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 [7],

C. Yu, W. Brown and M. Ciesielski are with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, 01003, USA (ycunxi@umass.edu, webrown@umass.edu, ciesiel@ecs.umass.edu).

D. Liu is with Intel Corporation, Hudson, MA, 01749, USA.

A. Rossi is with LERIA lab, Computer Sciences department of University of Angers, 49045 Angers, France.

MiniSAT [8], and others. Some of them, such as CryptoMiniSAT [9], specifically target XOR-rich circuits, but, like all others, are based on a computationally expensive Davis/Putnam/Logemann/Loveland (DPLL) decision procedure. Several techniques combine linear arithmetic constraints with Boolean SAT in a unified algebraic domain [10]; or combine automatic test pattern generation (ATPG) and modular arithmetic constraint-solving techniques for the purpose of test generation and assertion checking [11]; but they do not offer sufficient scalability. Approaches based on ILP models of the arithmetic operators [12] [13] 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 [14]. Some of the most effective SMT solvers, potentially applicable to our problem, are Boolector [15], Z3 [16], and CVC [17]. However, SMT solvers still model the problem as a decision problem and, as demonstrated by our experimental results, are not efficient at solving verification problems that appear in arithmetic circuits.

C. Theorem provers

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 [18], PVS [19], Boyer-Moore/ACL2 [20], and Nqthm [18][21]. These systems are characterized by high abstraction and powerful logic expressiveness, but they are highly interactive, require domain knowledge, extensive user guidance, and expertise for efficient use. 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 [22] or [23], are incomplete and "may fail to generate the proof because additional lemmas are needed" [23].

D. Computer Algebra Methods

One of the most advanced techniques that have potential to solve the arithmetic verification problem are those based on symbolic Computer Algebra [24]. These methods model the arithmetic circuit specification and its hardware implementation as polynomials [25],[26],[27],[28],[29],[30]. The verification goal is to prove that 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 components that implement the circuit. For example, the specification of a multiplier circuit with word-level inputs X, Y and output Z is $F = Z - X \cdot Y$. The implementation polynomials are derived from gate equations, similar to those shown later in Equation(1).

To systematically perform polynomial division, term ordering ">" is imposed on monomials, so that each polynomial has a well defined leading term *lt()*. If polynomial f contains some term t that is divisible by the leading term $lt(g)$ of polynomial g, then the division of f by g gives a remainder polynomial $r = f - \frac{t}{lt(g)} \cdot g$. In this case, we say that f *reduces* to r modulo g, denoted $f \xrightarrow{g} r$. With this, the verification problem is posed as the reduction of F modulo B, denoted $F \xrightarrow{B}_+ r$. The remainder r has the property that no term in r is divisible by the leading term of any polynomial f_i in B. The set of polynomials $B = \{f_1, \ldots, f_s\}$ generates an *ideal* $J = \langle f_1, ..., f_s\rangle$ with $f_i \in \mathbb{Z}[X]$, defined as: $J = \langle f_1, ..., f_s\rangle = h_1 f_1 + ... + h_s f_s$, with h_i from polynomial ring R. The polynomials $f_1, ..., f_s$ are called the bases, or *generators*, of the ideal J. In the context of circuit verification, they model the *implementation* of the circuit.

Given a set $f_1, ..., f_s$ of generators of J, a set of all simultaneous solutions to a system of equations $f_1(x_1, ..., x_n) = 0; ..., f_s(x_1, ..., x_n) = 0$ is called a *variety* $V(J)$. Verification problem is then formulated as testing if the specification F vanishes on $V(J)$[1] [29] [30]. In some cases, this test can be simplified to checking if $F \in J$, which is known in computer algebra as *ideal membership* testing[2].

A standard procedure to test if $F \in J$ is to divide polynomial F by $f_1, ..., f_s$, one by one. The goal is to cancel, at each iteration, the leading term of F using one of the leading terms of $f_1, ..., f_s$. If the remainder of the division is $r = 0$, then F vanishes on $V(J)$, proving that 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 generators, $G = \{g_1, ..., g_t\}$, called *Groebner basis*. Without Groebner basis one cannot answer the question whether $F \in J$.

A number of algorithms have been developed to compute Groebner basis over the field, including the well known Buchberger's algorithm [31]. However this algorithm is computationally expensive, as it computes the so called *S-polynomials*, by performing expensive division operation on all pairs of polynomials in B. Even with newer algorithms, such as F4 [32], the computational complexity of Groebner basis computation remains prohibitively large for arithmetic circuits. Furthermore, what is the most important, these algorithm do not apply directly to rings over integers, $\mathbb{Z}_{2^n}$, which is needed to solve the verification problem for arithmetic circuits considered in this work. In general, this problem cannot be solved by testing if F is a member of an ideal $J = \langle f_1, ..., f_s\rangle$, i.e., if $F \in J$. Many of the results related to ideal membership that are valid over algebraically closed fields are fundamentally unsolved over integers $\mathbb{Z}$. It has been shown that solving the problem for $\mathbb{Z}_{2^n}$ requires testing if $F \in I(V(J))$, where $I(V(J))$ is a set of all polynomials that vanish on $V(J)$ [33] [29].

[1]Polynomial f is said to vanish on V if $\forall a \in V: \; f(a) = 0$.

[2]In general, one must test if $F \in I(V(J))$. Only for finite fields $\mathbb{F}_{2^q}$ that this test reduces to $F \in J$. Details can be found in [24] [29].

Wienand et. al. [27] model an arithmetic circuit as an *arithmetic bit-level* (ABL) network of adders and other arithmetic operators. Both the specification and the arithmetic 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 [34]. In [28], 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 values of variables x to (0,1). While mathematically elegant, adding this constraint for all variables makes the method computationally expensive for gate-level circuits. For this reason, the method of [28] is limited to ABL networks composed of half adders (HA).

Kalla, et. al [29][30][35][36], 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 [30], 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 works of Basith et. al. [37] and Ciesielski et. al. [38], 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 potential 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 [39], by computing input signature from the known output signature using a *network-flow approach*. This technique also relies on the half-adder (HA) 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 contrast, the technique described in this paper works on an arbitrary, unstructured gate-level arithmetic circuit without requiring any reference to higher level models such as adders; it can efficiently handle nonlinear circuits without a need to distinguish between linear and nonlinear parts.

In summary, the problem of formally verifying integer arithmetic circuits over integers $\mathbb{Z}_{2^n}$ remains open [40]. 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.

The technique proposed in this paper solves the functional verification problem by devising an alternative but equivalent method, based on polynomial substitution and elimination, initially described in [41]. It correctly implements ideal membership testing without a need for expensive division process with Groebner basis. The results demonstrate that it scales better and is more efficient than the state-of-the-art computer algebra methods.

III. Function Extraction

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 the encoding of the primary outputs (called the *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 identifies the arithmetic function implemented by the circuit.

The method uses an algebraic model of the circuit, with logic gates represented by algebraic expressions, while correctly modeling signals as Boolean variables. In contrast to [39], it works directly on unstructured, gate-level implementations. And, in contrast to [28],[30] 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 $\mathbb{Z}_{2^n}$[3]. It provides a practical method for checking if the implementation satisfies the specification without resorting to the ideal membership testing in $\mathbb{Z}_{2^n}$.

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 cell gates obtained by synthesis and technology mapping. In fact, the proposed model admits a hybrid network, composed of an arbitrary collection of logic gates and bit-level arithmetic components. At one extreme, it can be a purely gate-level circuit; at the other, a network composed of arithmetic components only. 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

[3]The functional abstraction technique described in [30] applies only to Galois field circuits and is based on polynomial reduction via Groebner basis.

modulo 2^n). The following algebraic equations are used to describe basic 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 (cf. Theorem 1 in Section III.C).

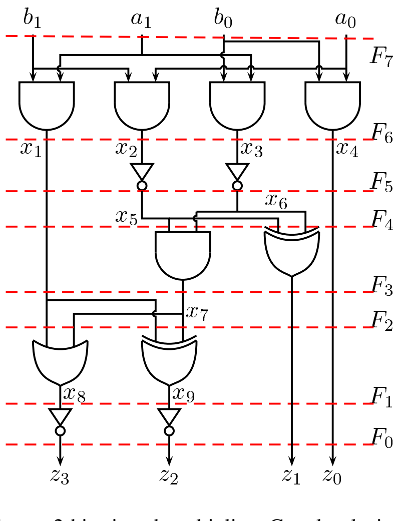


Fig. 1. Verifying a 2-bit signed multiplier: Gate-level circuit with output signature $Sig_{out} = -8z_3 + 4z_2 + 2z_1 + z_0$.

Algorithm 1 Verification Flow

```
Input: Gate-level netlist, output signature $Sig_{out}$ (input signature $Sig_{in}$)
Output: $Pseudo-Boolean$ expression extracted by rewriting
1: Parse gate-level netlist; create algebraic equations for gates/modules
2: Find ordering for variable substitution (levelization, dependency)
3: $i$ = 0; $F_i$ = $Sig_{out}$
4: while there are unused equations do
5:     Rewrite: $F_{i+1}$ = $F_i$ with variables substituted with gate equations;
6:     $i$ = i + 1
7: return $F$ = $F_i$ (to be compared with $Sig_{in}$)
```

B. Outline of the Approach

The proposed verification flow is outlined in Algorithm 1. The inputs to the algorithm are: the gate-level netlist (*implementation*); output signature Sig_{out} (*encoding* of the result a PO); and optionally the input signature Sig_{in} (*specification*). The first step is to translate the gate-level implementation into algebraic equations ($line$ 1). Then, the algebraic equations are ordered according to the circuit structure and its topology by algorithms that try to keep the size of the intermediate expressions small ($line$ 2). Specific algorithms (*levelization* and *dependency*) are discussed in the next section. The rewriting process is an iterative application of rewriting one pseudo-Boolean expression into another in the predetermined order ($lines$ $3-6$), starting with the output signature Sig_{out} at the primary outputs, PO. At each iteration, all variables in the current expression are substituted by the corresponding gate expressions. Each iteration produces its own expression, F_i ($line$ 5). The process ends when the rewriting reaches the primary inputs, PI, ($line$ 7), or when all equations have been used. The resulting expression F can then be compared with Sig_{in}, if it was provided by the designer, to determine if the circuit correctly implements the specification. Otherwise, the computed expression F determines the arithmetic function implemented by the circuit.

The rewriting process is illustrated with a simple 2-bit signed multiplier example, shown in Fig. 1. Each equation corresponds to a *cut* in the circuit, i.e., a set of signals that separate primary inputs from primary outputs; its pseudo-Boolean expression is denoted in the Figure by F_i. 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 expression $F_7 = 4a_1b_1 - 2a_0b_1 - 2a_1b_0 + a_0b_0$. Note the local increase in the polynomial size (at F_2 or F_4) known as "fat belly" effect, before it is 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 big influence on the size of the fat belly and the efficiency of the method. The following heuristics are used to keep the size of the intermediate expressions as small as possible.

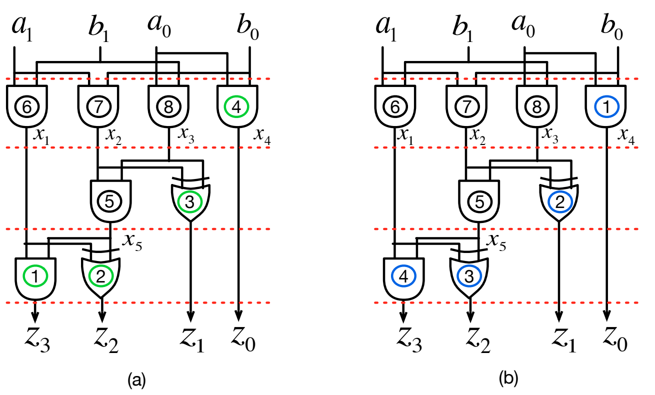


Fig. 2. Substitution order analysis using a 2-bit multiplier.

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

1) Substitution order: The substitution order has the greatest influence on the intermediate expression size (#. monomials). For a small difference between two orders, the maximum intermediate expression size may be several orders of magnitude larger in a large design. We illustrate the impact of the substitution order using a 2-bit multiplier (Figure 2). Orders (a) and (b) are two different substitution solutions which the first four iterations are different. We record the intermediate expression size step by step during rewriting (TABLE I). We can see order (b) identifies a larger peak than order (a). We present two methods to find the efficient substitution order: *Dependency* and *Levelization*.

Dependency: 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.

Levelization: To further increase the efficiency of substitution, a ”horizontal” constraint is also 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

#.iteration	1	2	3	4	5	6	7	8
Exp. size(a)	4	4	4	6	6	4	4	4
Exp. size(b)	4	4	6	8	6	4	4	4

TABLE I
2-BIT MULTIPLIER INTERMEDIA EXPRESSION SIZE OF TWO SUBSTITUTION SEQUENCE

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.

We demonstrate why substitution order greatly impacts the rewriting process using larger examples (Figure 3). We compare the rewriting process of 4-bit, 6-bit, and 8-bit CSA multipliers using *dependency* and *levelization*. In Figure 3, the *x*-axis represents the rewriting process in percentage of computation. The *y*-axis represents the size of intermediate expression, i.e. the number of monomials in the expressions. We can see that the difference of the size of the intermediate expression using *dependency* and *levelization* increases when the size of the design is increasing. This means that the substitution order has greater impact on the rewriting process if the designs are more complex.

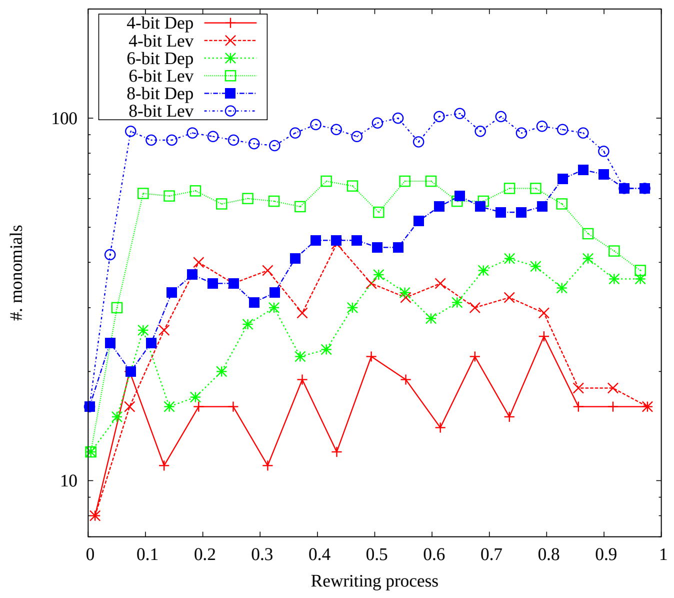


Fig. 3. Substitution order analysis using 4-bit, 6-bit, and 8-bit multiplier. *Dep* is dependency; *Lev* is levelization.

2) Fanouts: The size of the intermediate polynomial generated during rewriting can be reduced by identifying variables that depend on common inputs (fanouts of some variables). In this case, the substitution of such variables can be done simultaneously as this increases a chance for eliminating common subexpressions. For example, in Fig. 1 variables x_8, x_9 in subexpression $(8x_8 - 4x_9)$ of F_1 depend on common fanout variables x_1 and x_7. As a result, the subexpression $(8x_8 - 4x_9) = 4(2x_8 - x_9)$ reduces to $4(x_1 + x_7)$, without introducing a nonlinear term $8x_1x_7$, so F_1 can be directly transformed into F_3. Such nonlinear terms are particularly harmful if their variables continue to be substituted by other variables, potentially leading to an exponential explosion.

Another simplification that can be applied during rewriting relies on recognizing some pre-defined multiple-input modules with known I/O signatures, such as half adder or full adder. Adders are particularly useful, since they exhibit linear relationship between their inputs and outputs. For example, the circuit in Fig. 1 contains a half adder with inputs x_5, x_6 and outputs x_7, z_1, with a linear I/O relationship described by $(x_5+x_6 = 2x_7+z_1)$. In this case, the subexpression $4x_7+2z_1$ in F_3 can be directly translated into $2(x_5+x_6)$, avoiding an intermediate nonlinear term $4x_5x_6$ of F_4. As a result, cut F_3 can be directly transformed into F_5.

3) 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 ($Z = A^2$) 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.

4) 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 AOI, OAI, 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.

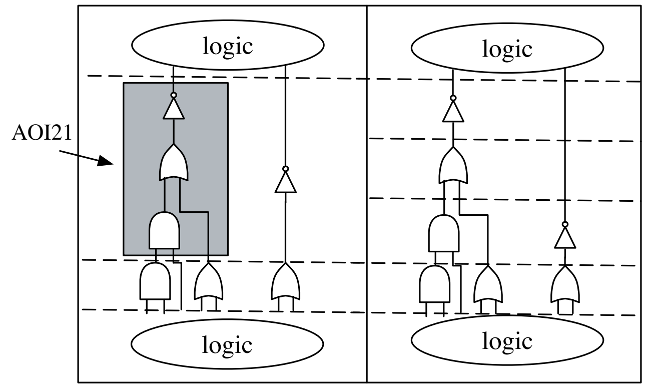


Fig. 4. Expanding complex gates for cut rewriting.

5) Binary signals: During elimination, the expensive division by the ideal $\langle x^2 - x\rangle$, employed by [28], 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 or x^2y, it will be replaced by xy. With this, an expression, such as $xyx - yxy$, will immediately reduce to 0. This significantly simplifies the procedure, compared to the division by $\langle x^2 - x\rangle$. This approach has also been used in recent works [29], [30] that targeted GF circuits. The lack of this kind of simplification was the main reason for the existence of *residual expression* in earlier works that advocated algebraic approach [37].

6) 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*.

It is essential to guarantee that the algebraic expressions of logic gates (eq. 1) correctly model Boolean signal variables. That is, the internal signal variables computed using those algebraic models must evaluate to exactly the same values as when using strictly Boolean methods, for all possible binary input combinations. With this, many potentially large algebraic subexpressions produced during the substitution will reduce to zero. This point can be illustrated with an example of the OR$_1$ gate with output x_{11} in the 3-bit adder in Fig. 5, now written in algebraic rather than Boolean form (Figure 6). As one can see, the value of x_{11} is exactly the same as the one obtained above using strictly Boolean methods (where x_5x_8 was also shown to reduce to 0).

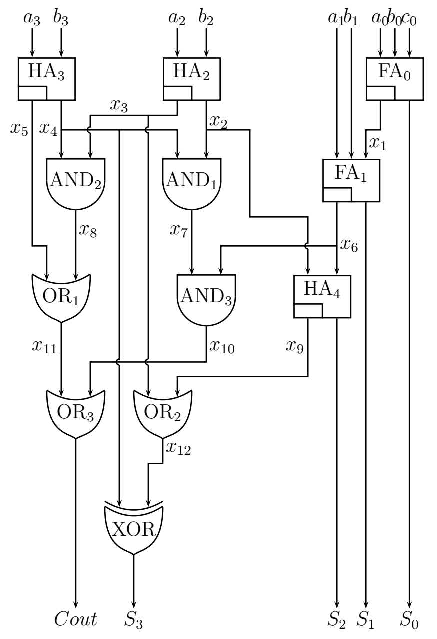


Fig. 5. Parallel prefix adder, hybrid model

$$\textbf{\textit{Algebraic}}\left\{\begin{array}{l} x_{11} = x_5 + x_8 - (x_5 x_8) \\ x_5 = a_3 b_3 \\ x_8 = a_3 + b_3 - 2a_3 b_3 \\ x_5 x_8 = (a_3^2 b_3 + a_3 b_3^2 - 2a_3^2 b_3^2) x_3 \\ \quad = (a_3 b_3 + a_3 b_3 - 2a_3 b_3) x_3 = 0 \\ x_{11} = x_5 + x_8 \end{array}\right.$$

$$\textbf{\textit{Boolean}}\left\{\begin{array}{l} x_5 x_8 \Rightarrow x_5 \wedge x_8 \\ = (a_3 \wedge b_3) \wedge [\,(\neg a_3 \wedge b_3) \vee (a_3 \wedge \neg b_3)\,] \\ = [\,(a_3 \wedge b_3) \wedge (\neg a_3 \wedge b_3)\,] \vee [\,(a_3 \wedge b_3) \wedge (a_3 \wedge \neg b_3)] \\ = [(a_3 \wedge \neg a_3 \wedge b_3)] \vee [(a_3 \wedge b_3 \wedge \neg b_3)] \\ = 0 \end{array}\right.$$

Fig. 6. Proof that x_5x_8 (in Figure 5) evaluates to 0 using both, the computer algebraic and Boolean methods.

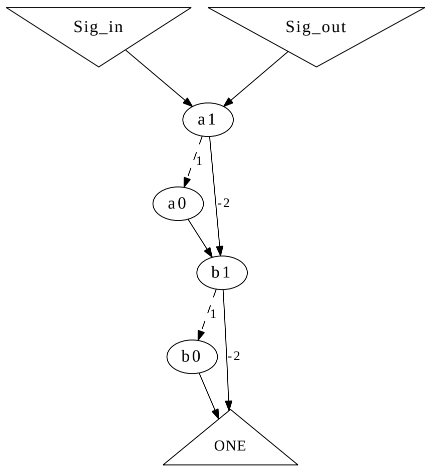


Fig. 7. Arithmetic function of a 2-bit multiplier extracted from the circuit using TED in normal factored form: $Sig_{in} = (-2a_1 + a_0)(-2b_1 + b_0)$.

C. Properties of Computed 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 [2] or TED [4] that can check equivalence between two word-level outputs expressed in bit-level inputs. In the 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.

If the specification is not given, TED can provide 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 the ordering of variables from which such a form can be obtained [42]. 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, 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. 7 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, $A \cdot B$, where 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.

Theorem 1: *Given a combinational circuit composed of basic logic gates, the 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; and 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 done by induction on i, the step when polynomial F_i is transformed into F_{i+1}. Base case: polynomial $F_0 = Sig_{out}$ is unique. Also, as discussed above, algebraic representation of each logic gate is unique.

Induction phase: Assuming that F_i is unique, we 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. $\square$

This theorem 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).

IV. Experimental Results

The verification technique described in this paper was implemented in C++. It performs rudimentary variable substitution and elimination, using the ordering strategy and implementation discussed in Section III-B. The program was tested on a number of gate-level combinational arithmetic circuits, taken from [43]: CSA multipliers, add-multiply, matrix multipliers, squaring, etc., with operands ranging from 64 to 512 bits. The results are shown in Tables III and IV. The experiments were conducted on a PC with Intel Processor Core i5-3470 CPU 3.20GHz x4 with 15.6 GB memory. The gate-level structures were obtained by direct translation of standard implementation of the designs onto basic logic gates [43]. The designs labeled with extension *.syn* were synthesized and mapped using ABC system [7] (commands: *strash; logic; map*) onto *mcnc.genlib* standard cell library. The plot for CPU runtime in Fig. 8 a) 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 [39] (col. 5) and the exponential time complexity of other tools.

As proposed in Section III-B, the reason why our technique is efficient is that rewriting the Sig_{out} provides significant internal expression elimination. We demonstrate this by measuring the size of the internal expressions of Sig_{out} and the individual output bit expressions (Figure 9). We can see that the expressions for z_5, z_6, z_7 are more than 100 times larger than the Sig_{out} in the middle of the rewriting process. However, each output bit expression contains many common monomials which can be eliminated by weighted addition (i.e. Sig_{out}).

A. SAT experiments

We tested the applicability of SAT tools to the the type of arithmetic verification problems described in this paper. The functional verification problem was modeled as a combinational equivalence checking problem, generated with a miter using ABC (command *miter*)[7], with the reference design generated by ABC using [*gen -N -m*] command. Then, we check if the miter is unsatisfiable. The state-of-the-art SAT solvers were tested using the CNF files created by ABC. ABC was also tested using the combinational equivalence checking *cec* (Table IV).

The *CEC* approach in ABC is based on AIG rewriting via structural hashing, simulation and the state-of-the-art SAT [44]. This technique reduces the overall complexity of checking equivalence between two designs by finding equivalent internal AIG nodes. However, finding internal equivalent nodes in non-linear arithmetic designs is very difficult. In Table II, N_1, N_2 are the numbers of AIG nodes before and after function reduction [$fraig - v$] [7]). Δ_1 shows the percentage of reduced nodes. The reference design is generated by ABC [*gen -N -m*] command. We can see that *fraig* is unable to identify and merge the internal equivalent nodes. Additionally, we evaluate the complexity of checking *Satisfiability* using SAT solver $lingeling$ [45]. N_3, N_4 are the numbers of clauses before and after simplification by [45]. Δ_2 shows the percentage of reduced clauses. We can see that both *fraig* and SAT solver cannot simplify the integer multiplier CEC problem. For this reason, such techniques are inefficient to verify non-linear arithmetic gate-level designs.

We also tested the SAT-based pseudo-Boolean solvers *MiniSat+* [46] and *PBSugar* [47] that have been applied to problems dealing with large pseudo-Boolean expressions. The specification is modeled as a pseudo-Boolean expression ($Sig_{out} - Sig_{in}$) and the gate-level implementation using the algebraic model, as in Eq. (1). If such constructed problem is *unSAT*, the implementation is bug-free. Both solvers successfully verified a 4-bit CSA multiplier, but were unable to solve the problem for a CSA multiplier circuit greater than six bits in 24 hours.

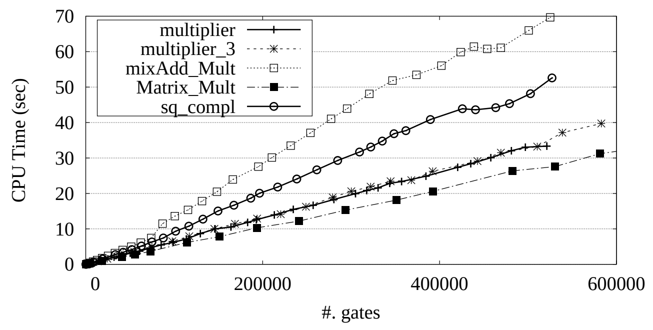
B. SMT experiments

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 modeling of the gate equations were tested:

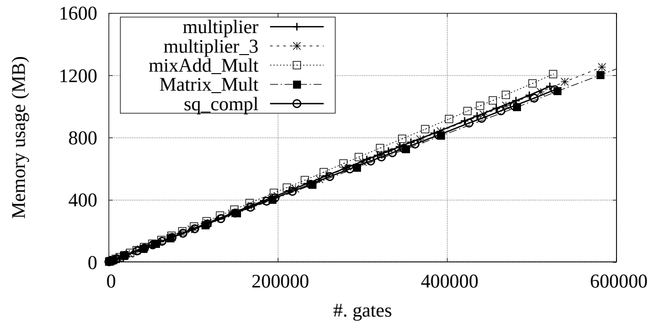
1) We directly translated the algebraic equations of the gate-level implementations into SMT2 format and modeled the specification (Sig_{out}-Sig_{in}) as a Pseudo-Boolean polynomial using Boolean vector operations.

2) The product circuit (miter) was translated directly into SAT by converting the CNF model into SMT2 format. The CNF files used in this experiment were the same as input to the SAT experiments. The second approach showed better performance; it is the one shown in Table IV.

Table IV gives comparison of our results for the synthesized multipliers with winners of recent SMT competitions and evaluation, including Boolector [15], Z3 [16], CVC4 [17]; minisat_blbd [48], lingeling [45] and the ABC system [7]; with the symbolic algebra tool, SINGULAR [34]; and Synopsys' *Formality* system. It shows that our technique surpasses those tools in CPU time by several orders of magnitude.



(a)



(b)

Fig. 8. Verifying combinational arithmetic circuits a) CPU time b) Memory usage

Size k		8-bit	16-bit	32-bit	64-bit	96-bit	128-bit
AIG	N_1	1173	5180	21641	88365	200140	356973
	N_2	1142	5140	21577	88278	200020	356822
	Δ_1	2.6%	0.7%	0.29%	0.09%	0.06%	0.08%
SAT	N_3	1655	7317	30543	124613	282159	503215
	N_4	1566	7133	30120	123758	280672	501512
	Δ_2	5.4%	2.5%	1.4%	0.07%	0.05%	0.03%

TABLE II
N_1, N_2 ARE #. NODES BEFORE AND AFTER *fraig -v* IN *ABC*; N_3, N_4 ARE #. CLAUSES BEFORE AND AFTER SIMPLIFICATION BY [45]

C. Limitations and Proposed Solutions

1) Circuit Boundaries: Currently, the described method of functional verification by signature rewriting requires knowledge of the I/O boundary of the circuit. Specifically, we need to know the output bits and their position (to be discussed in the next section), in order to generate the starting polynomial, Sig_{out}. We also need to know when to stop the rewriting process to correctly reason about the computed signature, Sig_{in}, and to determine if the circuit implements the expected arithmetic function. This seems to be a reasonable requirement for the functional *verification* of the given circuit, where the circuit has a well defined I/O boundary. However, if the method is used to *extract* an arithmetic function from a larger circuit, the exact I/O boundary may not be known. Presence of additional logic blocks at the inputs or outputs of the circuit clearly complicate the rewriting process. Future research will concentrate on relaxing the problem to the one with unknown I/O boundaries.

2) Output Encoding: As mentioned above, to obtain Sig_{out}, we need to know the correct encoding of the output bits. However, the encoding of the output bits may not be known. Hence, we propose a method by studying the intermediate expression of individual output bits to correctly assign the encoding position for the output bits.

The results shown in Figure 9 represent the intermediate expression size of individual output bit of a 4-bit multiplier. The horizontal axis represents the iteration number of rewriting process; the vertical axis represents the size of the expression at each point of the computation. We can see that the complexity of rewriting individual bits is different. The intermediate expression size of the 2^{nd} MSB is characterized by the highest complexity and LSB is the lowest one. The complexity of the individual output bits increases from z_0 to z_5. Based on this observation, we can determine the output encoding by monitoring the intermediate expression. The output bits close to MSB are very difficult to be extracted individually by our technique. The reason is that the intermediate expression is too large since there are few cancellations without the expressions from other outputs. However, to determine the output encoding, it is not necessary to rewrite the signature all the way to the primary inputs. The output encoding can be determined earlier in the process and the process will be terminated immediately. For example, in Figure 9, all the output bits can be recognized before iteration #35.

3) Effects of Synthesis on Function Extraction: The performance of our technique is sensitive to logic synthesis and technology mapping. In Figure 10, we compare the rewriting process with different logic synthesis techniques using 8-bit CSA Multiplier. The horizontal axis represents the rewriting process as percentage of the complete run; the vertical axis represents the size of the expression at each point of the computation. *Original* presents the rewriting process of 8-bit multiplier without any optimization. In Figure 10, curves *resyn* and *resyn3* are two different logic synthesis commands provided by ABC; curve *complex* refers to the mapping library includes the complex gates (e.g. AOI21, OAI221, etc.); curve "no-complex" refers to the library contains only 2 input logic gates.

We can see that the original 8-bit multiplier provides a much lower intermediate expression size, which means it is the easiest one to be verified. The synthesized multiplier mapped with complex gates are more difficult to verify than with the simple gates. The reason why the intermediate expression size is larger is that the logic synthesis technique and technology mapping techniques "restructure" the multiplier. This creates fewer possible cancellations in our rewriting technique. Using the heuristics proposed in Section III-B, we can verify a lightly-synthesized multiplier up to 256 (TABLE IV). However, the bit-optimized arithmetic design produced by $DesignCompiler$ or ABC dch remains challenge for this method.

V. CONCLUSIONS AND FUTURE WORK

The paper presents an efficient approach to derive the function computed by an integer arithmetic circuit from its gate-level implementation. It shows that such function extraction and the test if the implementation satisfies the specification can be efficiently implemented in algebraic domain using signature rewriting concept.

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, in form of a vanishing polynomial, may be useful in verification. Solving the verification problem for highly optimized bit-level circuits, synthesized with commercial tools, remains a challenge, as the synthesis tools are more aggressive in removing such redundancy. This, together with the need to know the circuit I/O boundary is currently the main limitation of this method.

We should also note that the verification of more structured circuits, containing larger pre-verified blocks will, in general, be easier. It requires fewer polynomials to be processed, which lowers the overall size of the problem, and there are fewer rewriting iterations. This is especially true if the relationship between the inputs and outputs of such a block is simpler than those of the internal gates. A simple example of such a structure, mentioned in Section III-B2 (*Fanouts*), is a half adder, whose I/O expressions can be reduced at its boundary to ($a + b = 2C + S$), without introducing any non-linear term. The early work on verification of arithmetic circuits mapped into a combination of half- and full adders and logic gates demonstrate an almost linear computational complexity

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)	445	0.01	1.5 MB	1.1K	0.05	3.5 MB	1.8K	0.19	6.4 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
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 III
CPU TIME AND MEMORY RESULTS (TO = TIMEOUT AFTER 3600 SEC; MO = MEMORY OUT OF 8 GB).

multiplier_synthesized												*multiplier_3*	
Statistics		Function-Extraction		[39]	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 IV
RESULTS FOR A SYNTHESIZED MULTIPLIER; COMPARISON WITH [39], SAT, SMT, AND COMMERCIAL TOOLS (TO = TIMEOUT AFTER 3600 SEC; UD = UNDECIDED; MO = MEMORY OUT OF 8 GB). *ABC WAS UNABLE TO SYNTHESIZE THE 512-BIT CSA MULTIPLIER DUE TO MEMORY LIMITATION.

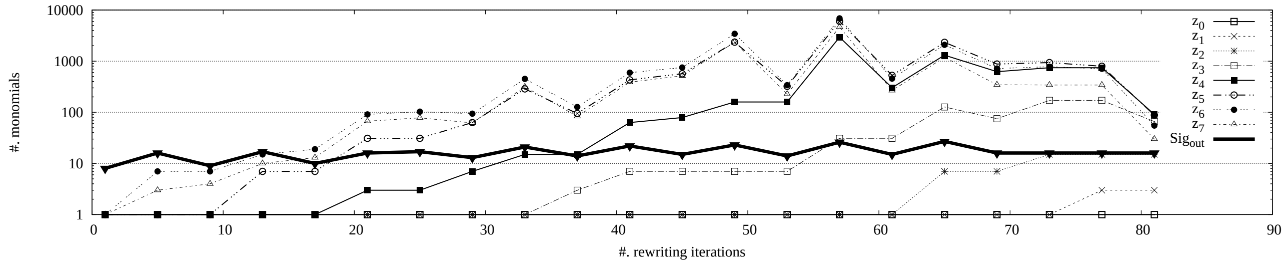


Fig. 9. Comparing rewriting of the expression Sig_{out} vs individual output bits for a 4-bit multiplier.

[37] [38]. However, as experimentally confirmed, sometimes the rewriting process benefits from breaking the aggregated complex gates into smaller ones to increase the chance of term cancellation during rewriting, c.f. Section III-B4 (*Complex gates*).

The verification method described in this paper can also be used to verify and debug *faulty circuits*. The basic idea is to perform signature rewriting in both direction, i.e., forward (from PI to PO) and backward (from PO to PI), up to some internal point (cut) of the circuit suspected of a bug. By comparing the two signatures one can determine if the circuit is correct of faulty. In the correct circuit, the two signature should be the same; if they are not, the difference between the two signatures can be used to identify the source of the bug. The details of this approach, for the case when the bug is caused by using a wrong type of logic gate (for example OR instead of AND) has been described in [49] and [50]. Initial results published in those papers, demonstrate applicability of this approach to large multipliers. The difficulty lies in deciding at which point the two signature should be compared, i.e., in efficient generation of cuts to locate a bug.

Future work will concentrate on verifying circuits synthesized with commercial tools; on verifying sequential and floating point circuits; and on arithmetic circuit debugging based on the signature rewriting concept.

ACKNOWLEDGMENT

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

REFERENCES

[1] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.
[2] R. E. Bryant and Y.-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," in *Design Automation Conference*, 1995, pp. 535–541.

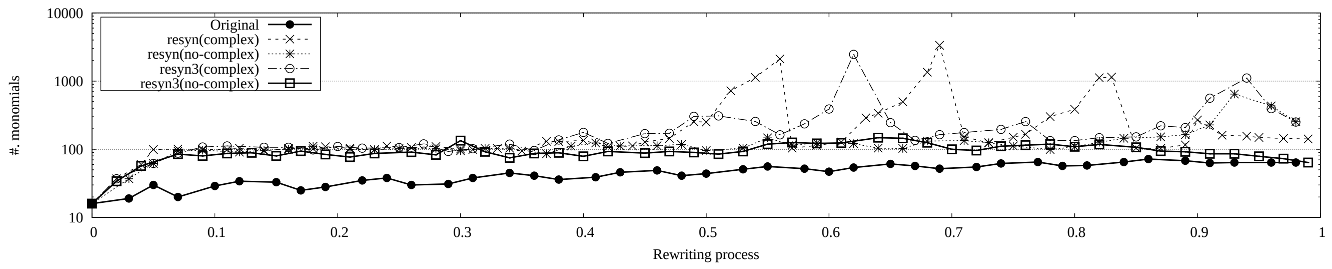


Fig. 10. Synthesis impacts on function extraction

[3] Y.-A. Chen and R. Bryant, "*PHDD: An Efficient Graph Representation for Floating Point Circuit Verification," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-97-134, 1997.

[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] F. Farahmandi, B. Alizadeh, and Z. Navabi, "Effective combination of algebraic techniques and decision diagrams to formally verify large arithmetic circuits," in *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on*. IEEE, 2014, pp. 338–343.

[6] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, E. R. V. Frolov, and A. Naik., "Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation," in *Computer Aided Verification (CAV)*. Springer, 2009, pp. 414–429.

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

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

[9] M. Soos, "Enhanced Gaussian Elimination in DPLL-based SAT Solvers." in *POS@ SAT*, 2010, pp. 2–14.

[10] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for hdl models using linear programming and 3-satisfiability," in *Design Automation Conference (DAC)*. IEEE, 1998, pp. 528–533.

[11] 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.

[12] R. Brinkmann and R. Drechsler, "RTL-datapath Verification using Integer Linear Programming," in *Proceedings of the 2002 Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE Computer Society, 2002, p. 741.

[13] Z. Zeng, K. R. Talupuru, and M. Ciesielski, "Functional Test Generation Based on Word-level sat," *Journal of Systems Architecture*, vol. 51, no. 8, pp. 488–511, 2005.

[14] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. ios press, 2009, vol. 185.

[15] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, 2015.

[16] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[17] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer aided verification (CAV)*. Springer, 2011, pp. 171–177.

[18] M. J. Gordon and T. F. Melham, "Introduction to HOL A Theorem Proving Environment for Higher Order Logic," in *Cambridge University Press*, 1993.

[19] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in *Automated Deduction - CADE-11*. Springer, 1992, pp. 748–752.

[20] B. Brock, M. Kaufmann, and J. S. Moore, "Acl2 theorems about commercial microprocessors," in *Formal Methods in Computer-Aided Design (FMCAD)*. Springer, 1996, pp. 275–293.

[21] M. Kaufmann and J. S. Moore, "Acl2: An industrial strength version of nqthm," in *COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on Computer Assurance*. IEEE, 1996, pp. 23–34.

[22] 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.

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

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

[25] T. Raudvere, A. K. Singh, I. Sander, and A. Jantsch, "System Level Verification of Digital Signal Processing Applications based on the Polynomial Abstraction Technique," in *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design (ICCAD)*. IEEE Computer Society, 2005, pp. 285–290.

[26] 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.

[27] 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.

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

[29] J. Lv, P. Kalla, and F. Enescu, "Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits," *IEEE Trans. on CAD*, vol. 32, no. 9, pp. 1409–1420, September 2013.

[30] 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 (DAC)*, 2014, pp. 1–6.

[31] B. Buchberger, "Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal," Ph.D. dissertation, Univ. Innsbruck, 1965.

[32] J.-C. Faugere, "A New Efficient Algorithm for Computing Groebner Bases (F4)," *Journal of Pure and Applied Algebra*, vol. 139, no. 1âĂŞ3, pp. 61 – 88, 1999.

[33] W. Adams and P. Loustanau, *An Introduction to Groebner Bases*. American Mathematical Society, 1994.

[34] 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.

[35] T. Pruss, P. Kalla, and F. Enescu, "Efficient Symbolic Computation for Word-level Abstraction from Combinational Circuits for Verification Over Finite Fields," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, p. 1, November 2015.

[36] X. Sun, P. Kalla, T. Pruss, and F. Enescu, "Formal verification of sequential galois field arithmetic circuits using algebraic geometry," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. EDA Consortium, 2015, pp. 1623–1628.

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

[38] M. Ciesielski and A. R. W. Brown, "Arithmetic Bit-level Verification using Network Flow Model," in *Haifa Verification Conference, HVC'13*. Springer, LNCS 8244, Nov. 2013, pp. 327–343.

[39] M. Ciesielski, W. Brown, D. Liu, and A. Rossi, "Function extraction from arithmetic bit-level circuits," in *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on*. IEEE, 2014, pp. 356–361.

[40] D. Cox, "Private Communication," Amherst College, February 2014.

[41] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of Gate-level Arithmetic Circuits by Function Extraction," in *52nd DAC*. ACM, 2015, pp. 52–57.

[42] 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.

[43] I. Koren, *Computer Arithmetic Algorithms*. Universities Press, 2002.

[44] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "Fraigs: A unifying representation for logic synthesis and verification," ERL Technical Report, Tech. Rep., 2005.

[45] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013," *Proceedings of SAT Competition*, pp. 51–52, 2013.

[46] N. Sörensson and N. Eén, "MiniSat 2.1 and MiniSat++ 1.0 - SAT race 2008 editions," *SAT*, p. 31, 2009.

[47] T. Naoyuki, S. Takehide, and B. Mutsunori, "PBSugar: A SAT-based Pseudo-Boolean Solver," *http://bach.istc.kobe-u.ac.jp/pbsugar*, 2013.

[48] A. Belov, D. Diepold, M. J. Heule, and M. Järvisalo, "SAT Competition 2014," 2014.

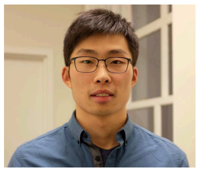
[49] S. Ghandali, C. Yu, D. Liu, B. Walter, , and M. Ciesielski, "Logic Debugging of Arithmetic Circuits," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2015, pp. 113–118.

[50] F. Farahmandi and P. Mishra, "Automated Test Generation for Debugging Arithmetic Circuits," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*. EDA Consortium, 2016.

Cunxi Yu received B.S. from Department of Information and Electronic Engineering, Zhejiang University City College, Hangzhou, China, in 2013. He is currently pursuing Ph.D degree in Department of Electrical and Computer Engineering, University of Massachusetts, Amherst.

His current research interests include formal verification, hardware security and logic synthesis. He is involved in formal analysis of integer and field arithmetic circuits. He is also involved in reverse engineering of camouflaged circuits using formal methods. He was research intern in IBM T.J Watson Research Center, Yorktown Height, NY, in 2015 and 2016.

Walter Brown is pursuing B.S. in Department of Electrical and Computer Engineering, University of Massachusetts, Amherst. He was REU student in summer 2013. He was awarded as Rising Researcher in University of Massachusetts, Amherst, in 2014. He is involved in formal verification and logic debugging of arithmetic circuits. He was summer intern in Cavium Inc, Marlborough, MA, in 2015.



Duo Liu received B.S. from Jiangnan University, Wuxi, China, in 2012. He received M.S. from Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, in 2015. He was summer intern in Synopsys Inc, Marlborough, MA, in summer 2015. He is now Verification Engineer with Intel Corporation, Hudson, MA, 01749.

André Rossi received an engineering degree in Electrical Engineering and a M.S in Production Sciences from INP Grenoble, France, in 2000, then a PhD degree from the same institution in 2003. After receiving the PhD award 'Prix de thèse' in 2005 from INP Grenoble, he served as an Associate Professor in Université de Bretagne-Sud, in Lorient, France, doing research in electronic design optimization. Since 2015, he is Professor in the computer sciences department of University of Angers, France.

Maciej Ciesielski Maciej Ciesielski is Professor in the Department of Electrical and Computer Engineering (ECE) at the University of Massachusetts, Amherst. He received M.S. in Electrical Engineering from Warsaw Technical University, Poland in 1974, and Ph.D. in Electrical Engineering from the University of Rochester, NY in 1983. From 1983 to 1986 he worked at GTE Laboratories on the SILC silicon compiler project. He joined the University of Massachusetts in 1987. He teaches and conducts research in the area of electronic design automation, and specifically in synthesis, simulation and formal verification of VLSI circuits and systems. In 2008 he received Doctorate Honoris Causa from the Universit*é* de Bretagne Sud, Lorient, France, for his contributions to the development of EDA tools for high level synthesis. He is author of over 120 publications in international conferences, archived journals and invited book chapters.