Formal Verification of Integer Multipliers by Combining Gröbner Basis with Logic Reduction

Amr Sayed-Ahmed[1] Daniel Große[1,2] Ulrich Kühne[1] Mathias Soeken[1,3] Rolf Drechsler[1,2]
[1]Faculty of Mathematics and Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany [3]Integrated Systems Laboratory (LSI), EPFL, Switzerland
{asahmed,grosse,ulrichk,msoeken,drechsle}@informatik.uni-bremen.de

***Abstract*—Formal verification utilizing symbolic computer algebra has demonstrated the ability to formally verify large Galois field arithmetic circuits and basic architectures of integer arithmetic circuits. The technique models the circuit as Gröbner basis polynomials and reduces the polynomial equation of the circuit specification wrt. the polynomials model. However, during the Gröbner basis reduction, the technique suffers from exponential blow-up in the size of the polynomials, if it is applied on parallel adders and recoded multipliers. In this paper, we address the reasons of this blow-up and present an approach that allows to apply the technique on basic and complex parallel architectures of multipliers. The approach is based on applying a logic reduction rule during Gröbner basis rewriting. The rule uses structural circuit information to remove terms that evaluate to zero before their blow-up. The experiments show that the approach is applicable up to 128 bit multipliers.**

I. Introduction

Verifying arithmetic circuits is hard. Since the famous FDIV bug in Intel's Pentium processor [1], a lot of effort has been spent developing automated and formal techniques which can prove the correctness of a design beyond mere testing. Among the basic operations, especially multiplication has turned out to be a tough nut to crack. Decision diagrams—such as BDDs or *BMDs—suffer from exponential blow-up. *Boolean Satisfiability* (SAT) techniques and also *Satisfiability Modulo Theories* (SMT) solvers fail to verify multiplier circuits of larger scale. Theorem provers can be used but require an enormous amount of manual effort and expert knowledge.

The most successful techniques up to today are based on reverse engineering an *arithmetic bit-level* (ABL) representation of the circuit [2] and—more recently—using computer algebra techniques on polynomial representations [3]–[8]. The latter techniques reduce the verification problem to membership testing of the specification polynomial in the ideal spanned by the circuit polynomials. The foundation for these techniques is Gröbner basis reduction. While these algebraic techniques have been applied successfully on large Galois Field arithmetic circuits [5] and ABL networks [3], [4], the verification of integer arithmetic on the gate netlist is limited by the exponential blow-up of the polynomial representation during Gröbner basis reduction.

The work in [7] improves the scalability by rewriting the polynomial model of the circuit, making the verification of a limited class of integer multipliers feasible. The technique allows the early cancellation of shared subterms in the polynomial representation, which effectively prevents the blow-up during Gröbner basis reduction. However, the technique does not scale for multipliers using parallel architectures such as *parallel prefix adders* (PPAs) or Booth recoding. The main reason—as identified by us—is the accumulation of *vanishing monomials*, which refers to monomials that always evaluate to zero. The problem is that these vanishing monomials cannot be identified locally using the approach of [7].

In this work, we present an algebraic technique that enables the verification of a large class of multiplier circuits, i.e., including basic and parallel multiplier architectures. Based on the observation of accumulating vanishing monomials, we propose a novel rewriting scheme. In particular, the technique makes use of structural knowledge on the circuit netlist in order to identify vanishing monomials early in the Gröbner basis reduction process. Using our technique, we can verify complex multiplier circuits of up to 128 bit in practical time.

The contributions of this work are:

1) Determining the reason of the inefficiency of applying the computer algebra technique using state-of-the-art algorithms to integer multipliers consisting of Booth partial products and PPAs.
2) Observing that rewriting as an explicit step in the membership testing algorithm is capable of circumventing blow-ups in the Gröbner basis reduction.
3) Proposing rewriting schemes based on logic reduction to remove vanishing monomials that appear in the reduction when verifying complex parallel integer multipliers.

II. Preliminaries

Verification using computer algebra is based on modeling the circuit under verification as Gröbner basis polynomials $G = \{g_1, \ldots, g_s\}$ and testing the membership of the specification polynomial p_{spec} in the Gröbner basis polynomials G. The membership testing is done by reducing (dividing) p_{spec} wrt. G. In the following, we define common notation and definitions and the ideal membership based on [9]. We explain the membership testing algorithm as implemented by [6], [7] to verify large basic integer arithmetic circuits.

A. Notation and Definitions

For a polynomial ring $K[x_1, \ldots, x_n]$ of n variables, a *monomial* $M = x_1^{\alpha_1} \cdot \cdots \cdot x_n^{\alpha_n}$ is the power product over the variables $x_1, \ldots, x_n$, where $\alpha_i \geq 0$. A *polynomial* $p = c_1 M_1 + \cdots + c_t M_t$ is a finite sum of terms, where each term is the product of a coefficient c_i and a monomial M_i. The monomials of a polynomial are ordered according to a

This work was supported in part by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1, by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative and by the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E.

monomial ordering '$>$', such that $M_1 > \cdots > M_t$, the *leading term* of the polynomial is $\mathrm{lt}(p) = c_1 M_1$, the *leading monomial* is $\mathrm{lm}(p) = M_1$, and the *leading coefficient* is $\mathrm{lc}(p) = c_1$. The set of variables appearing in polynomial p is denoted by $\mathrm{Vars}(p)$.

For a set of polynomials $P = \{p_1, \ldots, p_s\} \in K[x_1, \ldots, x_n]$, an *affine variety* $V(p_1, \ldots, p_s)$ is the set of all solutions of the polynomial equations $p_1(x_1, \ldots, x_n) = \cdots = p_s(x_1, \ldots, x_n) = 0$. An *ideal* $I = \langle P \rangle = \{\sum_{i=1}^{s} h_i \cdot p_i : h_i \in K[x_1, \ldots, x_n]\}$ is generated by this set of polynomials P, and we call P the basis (generators) of the ideal I. The ideal I may have many other bases. The bases are different representations of the set of polynomials P. One of these bases is called *Gröbner basis* $G = \{g_1, \ldots, g_{\hat{s}}\}$, for which $V(G) = V(I)$.

A polynomial reduction method named *S-polynomial* is designed to cancel the leading terms of two polynomials and is used by algorithms to reduce or divide Gröbner basis polynomials.

Definition 1: The *S-polynomial* of polynomials p and g in a polynomial set P, is the combination $\mathrm{Spoly}(p, g) = \frac{L}{\mathrm{lt}(p)} p - \frac{L}{\mathrm{lt}(g)} g$, where L is the least common multiple $\mathrm{LCM}(\mathrm{lm}(p), \mathrm{lm}(g))$. Note that $\mathrm{Spoly}(p, g)$ cancels the leading terms of p and g, the remainder r obtained in $\mathrm{Spoly}(p, g) \xrightarrow{P}_{+} r$ gives a new leading term.

To compute the Gröbner basis $G = \{g_1, ..., g_{\hat{s}}\}$ for an ideal $I\langle p_1, \ldots, p_s \rangle$, *Buchberger's algorithm* constructs G in a finite number of steps by applying $\mathrm{Spoly}(p, g) \xrightarrow{G}_{+} r$ in every step. A Gröbner basis is computed if all $\mathrm{Spoly}(p, g) \xrightarrow{G}_{+} 0$.

Lemma 1: Given a finite set $G \in K[x_1, \ldots, x_n]$, suppose that we have $p, g \in G$ such that $\mathrm{LCM}(\mathrm{lm}(p), \mathrm{lm}(g)) = \mathrm{lm}(p) \cdot \mathrm{lm}(g)$. In other words, the leading monomials of p and g are relatively prime. Then $\mathrm{Spoly}(p, g) \xrightarrow{G}_{+} 0$ [9].

According to Lemma 1, a given polynomial set is a Gröbner basis, if the leading monomials of all polynomials in the set are relatively prime. By combining this lemma with the affine variety concept of an ideal, we define the Gröbner basis of an ideal as follows:

Definition 2: A finite subset $G = \{g_1, \ldots, g_s\}$ wrt. a monomial order of an ideal I is said to be a Gröbner basis of I if $V(G) = V(I)$ and all leading monomials in G are relatively prime.

A given ideal may have different Gröbner bases, where one basis can be reduced wrt. a monomial ordering to other bases. These bases can be reduced again to a canonical representation of the ideal that is called *reduced Gröbner basis*.

The ability of the Gröbner basis to reveal the properties of the ideal allows to solve the ideal membership problem in an algorithmic fashion. The *ideal membership* algorithm decides whether a given polynomial p lies in ideal $I = \langle p_1, \ldots, p_s \rangle$, by combining the Gröbner basis with a division algorithm. First, it finds a Gröbner basis G for I. Then it applies a division algorithm to check that the remainder r on dividing p by G is equal to zero. The division is denoted $p \xrightarrow{G}_{+} r$.

B. Membership Testing Algorithm

The computer algebra technique verifies a circuit based on the ideal membership algorithm using an algorithm called *membership testing* (MT) that contains of the following four steps:

1) Model the circuit as Gröbner basis polynomials $G = \{g_1, \ldots, g_s\}$ and the specification as a polynomial p_{spec}.
2) Rewrite the Gröbner basis G to a new Gröbner basis G_n that has less variables.
3) Reduce (divide) p_{spec} wrt. G_n, denoted $p_{\mathrm{spec}} \xrightarrow{G_n}_{+} r$, where r is the remainder of dividing p_{spec} by G_n. Repeat this step until no term in r is divisible by the leading term of any polynomial in G_n.
4) If $r = 0$, the circuit satisfies the specification, otherwise, a mismatch between the algebraic model G_n and the specification equation is announced.

The second step (rewriting) is not required for the soundness of the algorithm. However, as we show later in the paper, it is crucial for the application to large integer circuits. In the following, each step of the MT algorithm is explained in detail. We explain Step 3 before Step 2 since it is easier to follow.

Step 1: Modeling a circuit as Gröbner Basis: The MT algorithm uses an algebraic model of the circuit. Logic gates are modeled by polynomials and signals as Boolean variables. The polynomials of basic Boolean gates are

$$\begin{aligned} z = \neg a &\implies g := -z + 1 - a \\ z = a \wedge b &\implies g := -z + ab \\ z = a \vee b &\implies g := -z + a + b - ab \\ z = a \oplus b &\implies g := -z + a + b - 2ab. \end{aligned}$$

Each logic gate is modeled in a way that the gate output variable z is described in terms of the gate input variables a, b. The polynomial $x^2 - x$ should be added to the model for each variable to enforce the Boolean domain. In practice, the ideal polynomials $\langle x^2 - x \rangle$ are replaced by reducing x^k to x every time its degree becomes greater than one during any computational step. For example, the monomial $x_1^2 x_2^3 x_3$ is equal to $x_1 x_2 x_3$ in the Boolean domain.

By ordering each variable of the model according to its reverse topological level in the circuit, the generated polynomials satisfy Def. 2 by construction. Every polynomial will be of the form $p_i := x_i + \mathrm{tail}(p_i)$, where x_i is the gate's output variable and $\mathrm{tail}(p_i)$ are terms consisting of the gate's input variables, describing the function implemented by the gate. According to this polynomial form, all the leading monomials of the model will be relatively prime.

Example 1: Consider the full adder circuit implementing the function $s_i + 2c_i = a_i + b_i + c_{i-1}$ shown in Fig. 1. Its algebraic model is

$$\begin{array}{ll} g_1 := -c_i - x_4 x_3 + x_4 + x_3 & g_2 := -s_i - 2x_1 c_{i-1} + x_1 + c_{i-1} \\ g_3 := -x_4 + x_2 c_{i-1} & g_4 := -x_3 + a_i b_i \\ g_5 := -x_2 - a_i b_i + a_i + b_i & g_6 := -x_1 - 2a_i b_i + a_i + b_i \end{array}$$

The specification polynomial is $p_{\mathrm{spec}} := -2c_i - s_i + c_{i-1} + b_i + a_i$. Ordering the polynomial variables in the reverse topological order of the circuit yields $c_i > s_i > x_4 > x_3 > x_2 > x_1 > c_{i-1} > b_i > a_i$. Following this order, the leading monomials of all polynomials will be relatively prime. E.g., the leading monomial of g_1 is c_i, and it is prime relative to all other leading monomials. According to Def. 2, the extracted algebraic model is therefore a Gröbner basis.

Modeling the circuit directly as Gröbner basis polynomials avoids Buchberger's algorithm and makes it computationally feasible to verify large arithmetic circuits using the MT

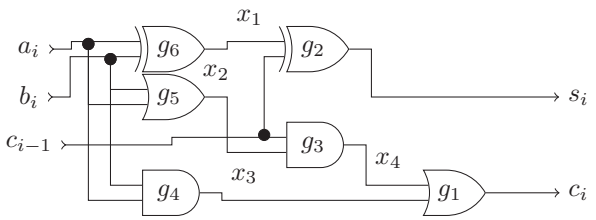

Fig. 1. A simple full adder.

Algorithm 1 Gröbner Basis Reduction (GB reduction)

Input: Specification polynomial p_{spec}, circuit polynomials $G = \{g_1, g_2, \ldots, g_s\}$
Output: Remainder of Gröbner basis reduction r

```
V ← OrderedPolynomialVariables(p_spec, G) /* Substitution ordering */
r ← p_spec
for i in 0 to |V| − 1 do
    if V[i] ∉ PrimaryInputs then
        Choose g_t ∈ G such that lm(g_t) = V[i]
        r ← Spoly(r, g_t)
    end if
end for
```

algorithm. In [5], it is applied on Galois field arithmetic. In case of integer arithmetic, the Gröbner basis reduction suffers from an exponential blow-up in the number of intermediate monomials during the division (reduction) process, because of nonlinear polynomial terms that model the carry chains. These nonlinear terms do not occur in case of Galois field arithmetic.

Step 3: Gröbner Basis Reduction: The reduction in Step 3 of the MT algorithm (in the remainder referred to as GB reduction) is performed according to Algorithm 1. Given a specification polynomial p_{spec} and a circuit model in form of a Gröbner basis G, p_{spec} is divided in every iteration by some polynomial $g \in G$ using the S-polynomial. The division can be seen as substituting the variables in p_{spec} with the corresponding tail terms of the respective polynomials in G. For example, given $p_{\text{spec}} := x_4x_3 + x_1$ and a polynomial $g := -x_4 + x_2x_1$, then $\text{Spoly}(p_{\text{spec}}, g) = \frac{x_4x_3}{x_4x_3}p_{\text{spec}} - \frac{x_4x_3}{-x_4}g = x_3x_2x_1 + x_1$, where the S-polynomial substitutes x_4 in p_{spec} with x_2x_1. The division (substitution) iterations are executed according to a certain order, the *substitution order*. This order is crucial in order to cancel the carry terms of integer arithmetic before the blow-up of the intermediate monomials. In [6], [7], the substitution ordering follows the reverse topological order of the circuit variables, in addition to the fanouts of the gates: Variables that have the same level and depend on common inputs (fanouts) must follow each other in the substitution.

Following Example 1, the extracted algebraic model is a Gröbner basis, therefore the GB reduction can be applied. As the full adder circuit has no gates with multiple fanouts, the substitution order will follow only the reverse topological order of the circuit:

$$p_{\text{spec}} \xrightarrow{g_1} -s_i + 2x_4x_3 - 2x_4 - 2x_3 + c_{i-1} + b_i + a_i$$
$$\xrightarrow{g_2} 2x_4x_3 - 2x_4 - 2x_3 + 2x_1c_{i-1} - x_1 + b_i + a_i$$
$$\xrightarrow{g_3} 2x_3x_2c_{i-1} - 2x_3 - 2x_2c_{i-1} + 2x_1c_{i-1} - x_1 + b_i + a_i$$
$$\xrightarrow{g_4} 2x_2c_{i-1}b_ia_i - 2x_2c_{i-1} + 2x_1c_{i-1} - x_1 - 2b_ia_i + b_i + a_i$$
$$\xrightarrow{g_5} 2x_1c_{i-1} - x_1 + 4c_{i-1}b_ia_i - 2c_{i-1}a_i - 2c_{i-1}b_i - 2a_ib_i + b_i + a_i \xrightarrow{g_6} 0$$

Step 2: Rewriting the Gröbner Basis: The bottleneck of the MT algorithm is the GB reduction which may cause a blow-up in the number of monomials in the remainder. Rewriting the Gröbner basis can avoid such blow-up, but typically depends on the considered types of circuits under verification. In [7], a rewriting scheme, called *fanout rewriting*, has been proposed based on the fanouts of the circuit gates, such that the model terms will depend only on shared variables. This dependency increases the chance of canceling common terms during the GB reduction. The rewriting is performed in two steps: 1) It finds the gates that have multiple fanouts and stores the corresponding output variables in a list, 2) It substitutes all variables that are not in this list, such that the model will depend only on fanouts, primary inputs, and primary outputs. This fanout rewriting and the substitution ordering of the GB reduction permit to cancel carry terms before their blow-up, as shown in the following example.

Example 2: Consider a 3-bit ripple carry adder implementing the function $\sum_{i=0}^{2} 2^i s_i = \sum_{i=0}^{2} 2^i(a_i + b_i)$. Since only the carry signals have multiple fanout, after fanout rewriting, all polynomials will depend on carry variables c_i, inputs, and outputs. The model is

$s_3 = c_2 \quad \Longrightarrow \; g_1 := -s_3 + c_2$
$c_2 = (a_2 \wedge b_2) \vee (a_2 \wedge c_1) \vee (b_2 \wedge c_1) \quad \Longrightarrow$
$g_2 := -c_2 \; \boxed{-2c_1b_2a_2 + c_1b_2 + c_1a_2 + b_2a_2}$
$s_2 = a_2 \oplus b_2 \oplus c_1 \quad \Longrightarrow$
$g_3 := -s_2 \; \boxed{+4c_1b_2a_2 - 2c_1b_2 - 2c_1a_2 - 2b_2a_2} + c_1 + b_2 + a_2$
$c_1 = (a_1 \wedge b_1) \vee (a_1 \wedge c_0) \vee (b_1 \wedge c_0) \quad \Longrightarrow$
$g_4 := -c_1 \; \boxed{-2c_0b_1a_1 + c_0b_1 + c_0a_1 + b_1a_1}$
$s_1 = a_1 \oplus b_1 \oplus c_0 \quad \Longrightarrow$
$g_5 := -s_1 \; \boxed{+4c_0b_1a_1 - 2c_0b_1 - 2c_0a_1 - 2b_1a_1} + c_0 + b_1 + a_1$
$c_0 = a_0 \wedge b_0 \quad \Longrightarrow \; g_6 := -c_0 \; \boxed{+b_0a_0}$
$s_0 = a_0 \oplus b_0 \quad \Longrightarrow \; g_7 := -s_0 \; \boxed{-2b_0a_0} + b_0 + a_0$

The specification polynomial is $p_{\text{spec}} := -8s_3 - 4s_2 - 2s_1 - s_0 + 4b_2 + 4a_2 + 2b_1 + 2a_1 + b_0 + a_0$. Rewriting the model wrt. circuit fanouts yields that polynomials g_2, g_3 have common monomials (colored green/dashed box in the example). These carry monomials cancel each other during the reduction of p_{spec} wrt. the rewritten model following the substitution order $s_3 > c_2 > s_2 > c_1 > s_1 > c_0 > s_0 > b_2 > b_1 > b_0 > a_2 > a_1 > a_0$. Similar cancellation occurs for equally colored terms of polynomials g_4, g_5 and polynomials g_6, g_7, respectively. Without rewriting, these carry terms will only be eliminated after reducing them to the input variables, which leads to an exponential increase in the number and size of monomials. Therefore, rewriting is required to enable verification for large integer circuits.

The simple rewriting described above is effective for the verification of basic multiplier architectures, i.e., multipliers with simple partial products generators and ripple carry adders in the last addition stage, but it fails to verify more complex architectures. In the next section, we will provide an explanation of this limitation, which forms the basis of our improved verification technique.

III. Problem Statement

If the membership testing algorithm with the existing rewriting schemes of Section II is used, the verification of integer multipliers that consist of parallel adders or Booth recoding

is unfeasible. The main reason are *vanishing monomials* (monomials that always evaluate to zero) which appear in every algebraic model of these complex multipliers. Unfortunately, the GB reduction cannot cancel these vanishing monomials before substituting them with input variables. Some of the vanishing monomials have the property that representing them by input variables will increase the number of intermediate monomials exponentially, therefore making the computation unfeasible. In this and the following section, we illustrate the vanishing monomials limitation with two examples: a parallel adder and a Booth partial product cell; and show how to overcome this problem by a new rewriting scheme enhanced by logic reduction.

Example 3: Consider a circuit model of a 3-bit PPA:[1]

$s_3 = c_2 \implies g_1 := -s_3 + c_2$

$c_2 = D_2 \vee (X_2 \wedge D_1) \vee (X_2 \wedge X_1 \wedge D_0) \implies g_2 := -c_2 + X_2D_2X_1D_1D_0 - X_2X_1D_1D_0 - X_2D_2X_1D_0 - X_2D_2D_1 + X_2X_1D_0 + X_2D_1 + D_2$

$s_2 = X_2 \oplus c_1 \implies g_3 := -s_2 - 2c_1X_2 + c1 + X_2$
$c_1 = D_1 \vee (X_1 \wedge D_0) \implies g_4 := -c_1 - X_1D_1D_0 + X_1D_0 + D_1$
$s_1 = X_1 \oplus c_0 \implies g_5 := -s_1 - 2c_0X_1 + c_0 + X_1$
$c_0 = D_0 \implies g_6 := -c_0 + D_0$
$s_0 = X_0 \implies g_7 := -s_0 + X_0$
$X_2 = a_2 \oplus b_2 \implies g_8 := -X_2 - 2b_2a_2 + b_2 + a_2$
$D_2 = a_2 \wedge b_2 \implies g_9 := -D_2 + b_2a_2$
$X_1 = a_1 \oplus b_1 \implies g_{10} := -X_1 - 2b_1a_1 + b_1 + a_1$
$D_1 = a_1 \wedge b_1 \implies g_{11} := -D_1 + b_1a_1$
$X_0 = a_0 \oplus b_0 \implies g_{12} := -X_0 - 2b_0a_0 + b_0 + a_0$
$D_0 = a_0 \wedge b_0 \implies g_{13} := -D_0 + b_0a_0.$

s_i is the sum bit, c_i is the carry bit, and for every input bits a_i, b_i, there is a generation bit D_i and a propagation bit X_i. The vanishing monomials in this model are colored red. As an example consider the vanishing monomial $X_1D_1D_0$ of polynomial g_4. Substituting X_1 and D_1 in this monomial yields $X_1D_1D_0 \xrightarrow{g_{10}} -2D_1D_0b_1a_1 + D_1D_0b_1 + D_1D_0a_1 \xrightarrow{g_{11}} -2D_0b_1a_1 + D_0b_1a_1 + D_0b_1a_1 = 0$. It is clear that the GB reduction can easily cancel this monomial. However, the corresponding monomials in the representation of the highest carry g_2 in an n-bit adder is $X_{n-1} \dots X_2X_1D_1D_0$. This follows from the circuit model $c_{n-1} = D_{n-1} \vee (X_{n-1} \wedge D_{n-2}) \vee (X_{n-1} \wedge X_{n-2} \wedge D_{n-3}) \vee \dots \vee (X_{n-1} \wedge X_{n-2} \wedge \dots \wedge X_2 \wedge D_1) \vee (X_{n-1} \wedge X_{n-2} \wedge \dots \wedge X_2 \wedge X_1 \wedge D_0)$.

By substituting in this monomial according to the order $X_{n-1} > D_{n-1} > \dots > X_0 > D_0$, the number of vanishing monomials will increase from 1 to 3^{n-1} monomials with a maximum size of $2n$ variables. Consider another vanishing monomial $X_2D_2X_1D_0$ of polynomial g_2, the corresponding vanishing monomial for the carry bit c_{n-1} is $X_{n-1}D_{n-1}X_{n-2} \dots X_2X_1D_0$. By substituting in this monomial with a different order $X_0 > D_0 > \dots > X_{n-1} > D_{n-1}$ compared to the previous one, the number of intermediate vanishing polynomials will increase to be about 3^{n-1} with a maximum size of $2n$ variables. From these two examples, we conclude that it is hard to find a substitution order to cancel all vanishing monomials before they blow up. The experimental results of [8] confirm the problem of the symbolic computer algebra approach with parallel adders. Their results show that the technique cannot verify Kogge-Stone adders with more 6 bits.

[1]Please recall that parallel prefix adders are typically found in the last stage of parallel multipliers.

Concluding our observations above, the core problem that we need to solve is the occurrence of a large number of vanishing monomials that lead to an exponential blow-up when reduced to the input variables.

IV. Logic Reduction Rewriting

This section describes the main contributions of our work. We present how to integrate logic reduction into Gröbner basis rewriting and present the two rewriting schemes that combine the elimination of vanishing monomials before they can cause a blow-up and the advantages given by the existing fanout rewriting technique.

A. Logic Reduction

To overcome the limitation caused by vanishing monomials during GB reduction, we propose to apply logic reduction during the rewriting of the Gröbner basis model (Step 2 of the membership testing algorithm in Section II-B), in order to remove vanishing monomials before their blow-up. Looking again at Example 3, it is easy to see that the monomial $X_1D_1D_0$ can be removed when considering that the variable X_1 is the XOR of a_1, b_1, and the variable D_1 is the AND of a_1, b_1. Based on this structural knowledge of the circuit model, we can conclude that the monomial always evaluates to zero since $(a \oplus b) \cdot (a \wedge b) = 0$ for all a and b. We refer to this as *XOR-AND vanishing rule*.

By keeping track of the original gate function and the input variables associated to each variable, we can effectively search for monomials that satisfy the XOR-AND vanishing rule. Applying this rule will remove all the vanishing monomials of the parallel adder model shown in Example 3, and will avoid the high computation cost of the GB reduction.

B. Rewriting Schemes

Although the correspondence of gates in the circuits to variables in the polynomials is given, the XOR-AND vanishing rule cannot directly be applied to the circuit obtained from Step 1 in the MT algorithm:

1) if *no* substitution is applied, one may not see monomials that contain both the XOR and AND of two common input variables, and
2) if *arbitrary* substitution is applied, internal XOR gates may be substituted.

Both cases prohibit the application of the XOR-AND vanishing rule. Consequently, substitution must be performed in an order that substitutes gates by preserving the monomials that represent XOR gates. Note that fanout rewriting does not preserve the XOR gates.

XOR rewriting: We propose an *XOR rewriting* scheme, carried out as step 2 in the MT algorithm, which performs the following steps:

1) Store all variables in a list V that refer to either input and output variables of an XOR gate or to primary inputs and primary outputs.
2) Rewrite the model using the S-polynomial method such that the model depends only on variables in V. After each substitution, apply the XOR-AND vanishing rule.

Common rewriting: As discussed in Section II-B, the fanout rewriting increases the chance that common subterms can be canceled during GB reduction. This positive effect is reduced by XOR rewriting, making the verification inefficient if only XOR rewriting is applied. Hence, we propose to carry out a further rewriting called *common rewriting*, which is similar to fanout rewriting, after XOR rewriting. Common rewriting simplifies the task of GB reduction by making the polynomials depend on shared variables. It rewrites the model obtained from XOR rewriting such that the polynomials depend only on variables that are used in more than one polynomial.[2]

Another part of the multiplier that shows the efficiency of the proposed XOR rewriting to reveal vanishing monomials is the Booth partial product cell. Although every cell has only one vanishing monomial, canceling it later by GB reduction causes a blow-up. Without early cancellation, every vanishing monomial will propagate through the carry chains of the multiplier and its size will increase by other variables. Because of that, canceling them by GB reduction will be computationally expensive.

C. Overall Algorithm

Both XOR rewriting and common rewriting follow two steps, which are identifying a set of variables and then substituting all remaining variables. Hence, the rewriting can be explained by a generalized algorithm, called *Gröbner Basis Rewriting* (GB-Rew), illustrated in Algorithm 2. It substitutes the variables that are not in V using the S-polynomial method. Additionally, in XOR rewriting, monomials are removed from the model using the XOR-AND vanishing rule after every substitution.

It considers the polynomials in reverse order of their leading monomial variables. For example, for a model of two polynomials $g_1 := x_1 + \mathrm{tail}(g_1)$ and $g_2 := x_2 + \mathrm{tail}(g_2)$ with monomial ordering[3] $x_2 > x_1$, the polynomial g_1 will be considered first.

The substitution order of the variables is chosen according to the number of terms in the tail part of their polynomials. For example, for two polynomials $g_1 := x_1 + \mathrm{tail}(g_1)$ and $g_2 := x_2 + \mathrm{tail}(g_2)$, variable x_1 is substituted before x_2 if the number of terms in $\mathrm{tail}(g_1)$ is smaller than the number of terms in $\mathrm{tail}(g_2)$.

After finishing the model rewriting and removing vanishing monomials, all polynomials whose leading monomial variables are not in the variables list V and which are not primary output variables will be removed, since the have been substituted during rewriting.

The overall rewriting scheme that is carried out as Step 2 in the MT algorithm is the sequential execution of XOR rewriting and common rewriting using the Gröbner basis rewriting. This is also illustrated by Algorithm 3, and is referred as *logic reduction rewriting*.

V. Experimental Results

The MT algorithm with logic reduction rewriting (MT-LR) and with fanout rewriting (MT-FO) [7] have been implemented in C++. The experiments have been carried out on an Intel(R) Core(TM) i5-3320M CPU (2.6 GHz, 16 GByte) running Linux. For the experiments, we generated different multiplier architectures using the online tool *Arithmetic Module Generator* [10]. The multipliers are given as Verilog RTL code. The designs were synthesized to gate level netlists using *Yosys* [11].

[2]This is very similar to fanout rewriting, but since we are no longer working on the original circuit model, one cannot strictly speak of fanout variables.

[3]Note that the variables are ordered according to the reverse topological order of the circuit, as explained in Section II.

Algorithm 2 Gröbner Basis Rewriting (GB-Rew)

Input: Variables V, Circuit Model G
Output: Model G_n rewritten wrt. V
1: **for** $g_i \in G$ **do** /* *in reverse order of leading monomials* */
2: $lv \leftarrow \mathrm{lm}(g_i)$
3: $r \leftarrow g_i - lv$
4: **while** $\mathrm{Vars}(r) \not\subseteq V$ **do**
5: Choose $v_t \in \mathrm{Vars}(r) \setminus V$
6: Choose $g_t \in G$ such that $\mathrm{lm}(g_t) = v_t$
7: $r \leftarrow \mathrm{Spoly}(r, g_t)$
8: $r \leftarrow \mathrm{XORAND\text{-}Rule}(r)$
9: **end while**
10: $g_i \leftarrow r + lv$
11: **end for**
12: $G_n \leftarrow \mathrm{UpdateModel}(G, V)$ /* *Remove polynomials whose leading terms are not in* V */
13: **return** G_n

Algorithm 3 Logic Reduction Rewriting

Input: Specification Polynomial p_{spec}, Circuit Model G
Output: Circuit Model G
1: $V \leftarrow \mathrm{XORRewritingVariables}(G)$
2: $G \leftarrow$ GB-Rew(V, G)
3: $V \leftarrow \mathrm{CommonRewritingVariables}(G)$
4: $G \leftarrow$ GB-Rew(V, G)
5: **return** G

To evaluate the practical time of the MT-LR algorithm in verifying multipliers with different architectures, we apply it to verify n-bit multipliers against the specification equation

$$\sum_{i=0}^{2n-1} -2^i s_i + \sum_{i=0}^{n-1} 2^i a_i \cdot \sum_{i=0}^{n-1} 2^i b_i \mod 2^{2n}.$$

This is done by dividing the algebraic module of multipliers wrt. the polynomial $p_{\mathrm{spec}} := \sum_{i=0}^{2n-1} -2^i s_i + \sum_{i=0}^{n-1} 2^i a_i \cdot \sum_{i=0}^{n-1} 2^i b_i$ and calculating $r \leftarrow r \bmod 2^{2n}$ by removing from the division remainder r the terms that have coefficients that are a multiple of 2^{2n}. Please note that we propose the idea of adding modulo 2^{2n} to the specification of integer multipliers, such that the specification matches multipliers that consist of Booth partial products or redundant binary addition trees. The remainder of dividing the specification equation without modulo 2^{2n} wrt. algebraic models of these multipliers is not equal to zero. The remainder has terms with coefficients that are multiple of 2^{2n}.

The multiplier architectures of benchmarks [10] are categorized according to 1) the type of the partial products generator, 2) the partial products accumulator, and 3) the last stage adder. In our experiments, we use two types of partial products generators, namely *simple partial products* (SP), and *Booth partial products* (BP) . The types of partial products accumulators are *array* (AR), *wallace tree* (WT), (4,2) *compressor tree* (CT), *redundant binary addition tree* (RT), and *dadda tree* (DT). Finally, the chosen types of the last stage adder are *ripple carry adder* (RC), *carry look-ahead adder* (CL), *brent-kung adder* (BK), *kogge-stone adder* (KS), and

TABLE I
VERIFICATION RESULTS FOR SIMPLE PARTIAL PRODUCTS MULTIPLIERS

Benchmark	I/O bits	Commercial (h:m:s)	CPP [13] (h:m:s)	MT-FO [7] (h:m:s)	MT-LR (h:m:s)
SP-AR-RC	16/32	00:00:01	00:01:23	00:00:01	00:00:02
SP-WT-CL	16/32	00:00:01	00:00:46	TO	00:00:05
SP-RT-KS	16/32	00:00:43	N/A	TO	00:00:17
SP-CT-BK	16/32	00:00:59	00:00:43	TO	00:00:04
SP-AR-RC	32/64	00:00:11	02:34:40	00:00:09	00:00:21
SP-WT-CL	32/64	00:00:06	00:15:12	TO	00:03:27
SP-DT-HC	32/64	00:00:09	N/A	TO	00:02:05
SP-CT-BK	32/64	TO	00:21:20	TO	00:01:35
SP-AR-RC	64/128	00:02:52	94:37:20	00:02:56	00:07:40
SP-WL-CL	64/128	00:00:36	05:46:40	TO	02:18:34
SP-RT-KS	64/128	TO	N/A	TO	02:51:12
SP-CT-BK	64/128	TO	05:31:44	TO	00:47:48
SP-AR-RC	128/256	01:03:34	TO	00:48:03	02:08:51
SP-CT-BK	128/256	TO	78:11:12	TO	14:03:33

TABLE II
VERIFICATION RESULTS FOR BOOTH PARTIAL PRODUCTS MULTIPLIERS

Benchmark	I/O bits	Commercial (h:m:s)	CPP [13] (h:m:s)	MT-FO [7] (h:m:s)	MT-LR (h:m:s)
BP-AR-RC	16/32	00:00:14	-	TO	00:00:02
BP-WT-CL	16/32	00:00:16	-	TO	00:00:09
BP-RT-KS	16/32	00:00:18	-	TO	00:00:17
BP-CT-BK	16/32	00:00:13	-	TO	00:00:06
BP-AR-RC	32/64	TO	-	TO	00:00:17
BP-WT-CL	32/64	TO	-	TO	00:04:46
BP-RT-KS	32/64	TO	-	TO	00:05:36
BP-CT-BK	32/64	TO	-	TO	00:02:20
BP-AR-RC	64/128	TO	-	TO	00:05:06
BP-WT-CL	64/128	TO	-	TO	03:03:48
BP-DT-HC	64/128	TO	-	TO	00:58:44
BP-CT-BK	64/128	TO	-	TO	00:37:53
BP-AR-RC	128/256	TO	-	TO	01:29:10
BP-CT-BK	128/256	TO	-	TO	15:14:49

hans-carlson adder (HC). In the following, the benchmarks are named according to their architecture features. Please note that all benchmarks time out after 100 hours when performing verification using a naive miter construction (one big miter; ABC [12] using command '*cec*').

In Tables I and II, we compare the run-times of the proposed algorithm MT-LR, against our re-implementation of MT-FO [7], a recent work combining recurrence relations and SAT-based equivalence checking [13] referred as *Checking Partial Product* (CPP) approach, and the equivalence checker of the commercial tool OneSpin (after enabling multiplier options). The first column of Tables I and II shows the name of the circuit. The second column gives the number of inputs and output bits. The next four columns provide the run-times. The time out (TO in the table) has been set to 100 hours. For the CPP approach, "-" refers to the fact that CPP can not be used for multipliers consisting of Booth partial products. Finally, N/A refers to not available in the respective paper. The experimental results clearly demonstrate the advantage of our approach. While for multipliers with simple partial products (Table I) the other approaches sometimes can verify the correctness, for the complex parallel architectures (Table II) only our approach solves the verification problem when the instances reach relevant sizes. As can be seen we are able to verify the correctness for up to 128 bits.

Table III shows some statistics about the MT-LR algorithm. The columns give the circuit name, number of circuit bits, number of vanishing monomial that are canceled by XOR-AND rule (#CVM), the run-time of the GB reduction after logic reduction rewriting, and finally statistics on the model after rewriting. The model statistics columns show number of polynomials (#P), number of monomials (#M), maximum size of a polynomial wrt. its number of monomials (#MP), and maximum size of a monomial wrt. its number of variables (#VM). The results of Table III show that multipliers with carry look ahead adders or with kogge-stone adder have the largest number of vanishing monomials and therefore the largest execution time. Moreover, it can be seen that the GB reduction spends a fraction of the total execution time of the MT-LR algorithm. Most of the time is spent in rewriting the circuit model by logic reduction rewriting.

TABLE III
STATISTICS FOR VERIFICATION OF MULTIPLIERS BY MT-LR

Benchmark	I/O bits	#CVM	GB reduction (h:m:s)	#P	#M	#MP	#VM
BP-WT-CL	32/64	39651	00:00:40	1965	18186	142	65
BP-RT-KS	32/64	42000	00:00:38	1989	23341	200	69
SP-DT-HC	32/64	15842	00:00:23	3011	18267	124	63
SP-CT-BK	32/64	4480	00:00:37	2702	37137	256	62
BP-WT-CL	64/128	325377	00:10:47	7180	71473	331	129
BP-DT-HC	64/128	134367	00:06:45	6491	70635	260	130
SP-RT-KS	64/128	290053	00:13:08	13106	95314	376	131
SP-CT-BK	64/128	22228	00:07:09	10676	148381	274	124
SP-CT-BK	128/256	106970	01:25:53	42016	592715	530	252

VI. CONCLUSION

In this paper we have presented a new algorithm which extends the computer algebra technique for verification of complex parallel multiplier architectures. The algorithm is based on canceling vanishing monomials in an efficient way before their blow-up during Gröbner basis reduction. Our approach rewrites the algebraic model of the multiplier based on the XOR gates of the multiplier netlist and searches for monomials that satisfy the XOR-AND vanishing rule. Experimental results demonstrated the efficiency of our approach, i.e., for all complex parallel multipliers we verified the correctness within seconds to 15 hours (for 128 bits), while all other approaches reached the timed out limit of 100 hours and gave no result.

REFERENCES

[1] H. P. Sharangpani and M. L. Barton, "Statistical analysis of floating point flaw in the pentium processor(1994)," Intel, Tech. Rep., Nov. 1994.

[2] E. Pavlenko, M. Wedler, D. Stoffel, O. Wienand, E. Karibaev, and W. Kunz, "Modeling of custom-designed arithmetic components in ABL normalization," in *FDL*, 2008, pp. 124–129.

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

[4] 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*, March 2011, pp. 1–6.

[5] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of galois field multipliers," in *DATE*, 2012, pp. 899–904.

[6] M. Ciesielski, C. Yu, D. Liu, and W. Brown, "Verification of gate-level arithmetic circuits by function extraction," in *DAC*, 2015, pp. 52:1–52:6.

[7] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.

[8] Y. Watanabe, N. Homma, T. Aoki, and T. Higuchi, "Application of symbolic computer algebra to arithmetic circuit verification," in *ICCD*, 2007, pp. 25–32.

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

[10] "Arithmetic module generator based on acg," available at http://www.aoki.ecei.tohoku.ac.jp/arith/, 2015.

[11] C. Wolf, "Yosys open synthesis suite," available at http://www.clifford.at/yosys/, 2015.

[12] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

[13] A. Sayed-Ahmed, U. Kühne, D. Große, and R. Drechsler, "Recurrence relations revisited: Scalable verification of bit level multiplier circuits," in *ISVLSI*, 2015, pp. 1–6.