ANALYSIS AND VERIFICATION OF ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA APPROACH

ANALYSIS AND VERIFICATION OF ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA APPROACH

A Dissertation Presented

by

TIANKAI SU

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2020

Electrical and Computer Engineering

ANALYSIS AND VERIFICATION OF ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA APPROACH

A Dissertation Presented

by

TIANKAI SU

Approved as to style and content by:

Maciej Ciesielski, Chair

George S. Avrunin, Member

Daniel Holcomb, Member

Weibo Gong, Member

Christopher V. Hollot, Department Head
Electrical and Computer Engineering

ABSTRACT

ANALYSIS AND VERIFICATION OF ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA APPROACH

FEBRUARY 2020

TIANKAI SU

B.Sc., NORTHEAST DIANLI UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Maciej Ciesielski

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 of efficient modeling of arithmetic circuits and data paths without resorting to computationally expensive Boolean methods, such as *Binary Decision Diagrams* (BDDs) and *Boolean Satisfiability* (SAT) that require "bit blasting", i.e., flattening the design to a bit-level netlist. Similarly, approaches that rely on computer algebra and *Satisfiability Modulo Theories* (SMT) methods are either too abstract to handle the bit-level complexity of arithmetic designs or require solving computationally expensive decision or satisfiability problems. On the other hand, theorem provers, popular solvers used in industry, require a significant human interaction and intimate knowledge of the design to guide the proof process.

The work proposed in this thesis aims at overcoming the limitations of verifying arithmetic circuits, especially at the post-synthesis, implementation phase. It ad-

dresses the verification problem at an algebraic level, treating an arithmetic circuit and its specification as an algebraic system. Specifically, verification approach employed in this work is based on the *algebraic rewriting* method of [86]. In this method, the circuit is modeled in the algebraic domain, where both the circuit specification and its gate-level implementation are represented as polynomials. This work formally analyzes the algebraic approach and compares it with the established computer algebra methods based on Gröbner basis reduction. It shows that algebraic rewriting is more effective than the Gröbner basis reduction from the computational point of view.

This thesis addresses two classes of arithmetic circuits that could not directly benefit from this type of functional verification, since performing algebraic rewriting of such circuits encounters a serious memory issue. The circuits that fall in the first category are approximate arithmetic circuits, such as truncated integer multipliers. Different truncation schemes are considered, including bit deletion, bit truncation, and rounding. The proposed verification method is based on reconstructing the truncated multiplier to a complete, exact multiplier; it is then followed by algebraic rewriting to prove that it indeed implements multiplication over the required range of bits. The reconstruction of the multiplier helps avoid the memory overload issue as it creates a "clean" multiplier with a well defined specification polynomial.

The other class of circuits that suffer from memory overload during algebraic rewriting are circuits subjected to some arithmetic constraints. An example of such circuits is a divider, where the divisor value cannot be zero. The other example can be found in the basic blocks of the constant divider, where the value of carry into each block must be less than the divisor value. In general, such constraints will be modeled using the concept of *vanishing monomials*. A case-splitting method is proposed along with the modified algebraic rewriting to resolve the memory issue. The proposed

verification method not only can prove that the circuit performs a correct function under the desired (valid) conditions, but also will test all the undesired (invalid) cases.

This work also addresses logic *debugging* of combinational arithmetic circuits over field $\mathbb{F}_{2^k}$, including Galois field multipliers. Galois Field (GF) arithmetic has numerous applications in digital communication, cryptography and security engineering, and formal verification of such circuits is of prime importance. In addition to functional verification of GF multipliers, this work proposes a novel and effective method for identifying and correcting bugs in such circuits, commonly referred to as *debugging*. In this work we propose a novel approach to debugging of GF arithmetic circuits based on *forward rewriting*, which enables functional verification and debugging at the same time. This technique can handle multiple bugs, does not suffer from the polynomial size explosion encountered by other methods, and allows one to identify and automatically correct bugs in GF circuits.

The techniques and algorithms proposed in this dissertation have been implemented in several computer programs, some stand-alone, and some integrated with a popular synthesis and verification tool, ABC [11]. The experimental results for verification and debugging are compared with the state-of-the-art SAT, SMT, and other computer algebraic solvers.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

With the ever-increasing size and complexity of integrated circuits (IC) and systems on chip (SoC), hardware verification has become a dominating factor of the overall design flow [32]. Particularly important and challenging is the verification of datapaths and their arithmetic components. 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.

Current formal verification techniques are ineffective when dealing with large arithmetic designs as they rely on the established Boolean techniques that require flattening the entire design into a bit-level netlist, informally referred to as "bit-blasting". In this work, we address this issue by solving the verification problem in algebraic domain instead of in strictly Boolean. The proposed technique is discussed in Chapter 3, is efficient and scalable to verify large standard arithmetic circuits, such as adders and multipliers. In Chapter 4 and Chapter 5, we further extend the technique to make it be able to handle some non-standard classes of arithmetic circuits. The issue of verification and logic debugging of Galois Field circuits has been addressed in Chapter 6 by a rewriting-based method.

In this chapter, we first review the hardware design flow and illustrate the importance of hardware verification in the flow. Then, traditional Boolean verification methods are briefly reviewed, along with their limitations. Those methods are not sufficiently effective to verify the large and complex circuits, especially modern arith-

metic circuits [10][16][24][58]. This provides the motivation of this thesis, which aims at overcoming the limitations of verifying large arithmetic circuits.

1.1 Hardware Design and Verification Flow

On May $26th, 1960$, the first planar integrated circuit was produced [84]. This event opened the door for *Integrated Circuits* (ICs) industry, and soon was followed by the *Electronic Design Automation* (EDA) [1][30]. During the next several decades, a large number of *Computer Aided Design* (CAD) tools have been developed to support design automation that would replace the handcrafted IC design process [54][33]. CAD software is widely used to increase the productivity of the semiconductor industry. Today, IC design and manufacturing gained maturity, owing to the *Very large-scale integration* (VLSI) design flow automation [49][80]. Current EDA and IC technology support designs with millions of logic gates and billions of individual transistors. A general IC design flow [44] and its basic steps are shown in Fig.1.1.

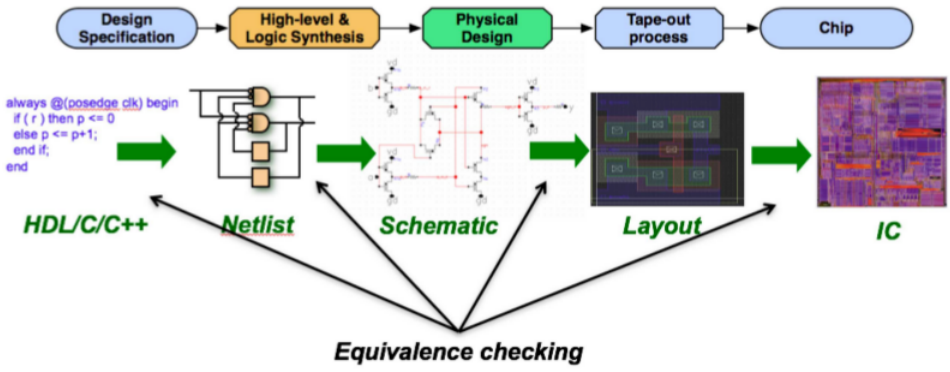


Figure 1.1: Typical industrial IC design flow.

The starting process for the design is a system-level specification. It is usually written in a software programming language (C, C++, etc.) or a behavioral Hardware Description Language (such as Verilog, System Verilog, VHDL) [23], which describes the behavior and functionality of the design. The specification is then compiled into a register-transfer-level (RTL) description and translated into Boolean expressions

of individual logic blocks and the interconnection logic [77]. After that, the generated logic expressions are synthesized and mapped onto gate-level netlists. Different optimization goals are used during logic synthesis, such as area, delay or power minimization, depending on the intended application and the target technology, ASIC [71] or FPGA [17][7]. Then, the layout design tools, commonly known as *Place and Route* [67][64], are used to perform physical design. Before the circuit is fabricated and taped out, it must be subjected to a thorough verification to guarantee its functional correctness.

Hardware verification is crucial and must be conducted during each step of the design process. Specifically, the goal of the verification is to check if the actual hardware implementation meets the required specification, that is, to ensure that no errors were introduced during any of the synthesis steps, either by the designer or by the CAD tools. In practice, it is common for designers to make manual, last-minute changes to a netlist, commonly known as *Engineering Change Orders* (*ECOs*) [2], and those can potentially introduce errors. As soon as one step of the design contains a bug, it has to be identified and corrected [48]. For this reason, the design has to be thoroughly verified at each step.

There are several types of verification, such as Equivalence Checking (*EC*) [61], Model Checking [79], Property checking [39], and Functional Verification [38]. Typically, equivalence checking is applied between different levels of abstraction of the design to check their equivalence before and after each optimization or transformation in the design flow. While simulation is one of the better understood and developed traditional approaches to verify a circuit, exhaustive simulation is not applicable to large modern designs, whose size continues to grow exponentially [89]. For this reason, several formal techniques have been developed to handle large practical circuits, including canonical decision diagrams (BDDs, BMDs, TEDs), Boolean Satisfiability

(SAT), Satisfiability Modulo Theories (SMT) and Theorem Proving. These techniques are briefly described in the next section.

This thesis focuses on functional verification of a particular class of designs, namely arithmetic circuits, which are harder to verify than logic circuits. In particular, we target gate-level implementation of combinational arithmetic circuits. The other kinds of hardware verification, such as model and property checking, physical verification, timing verification, clock domain crossing (CDC) verification, and other timing related verification are not the subject of this work.

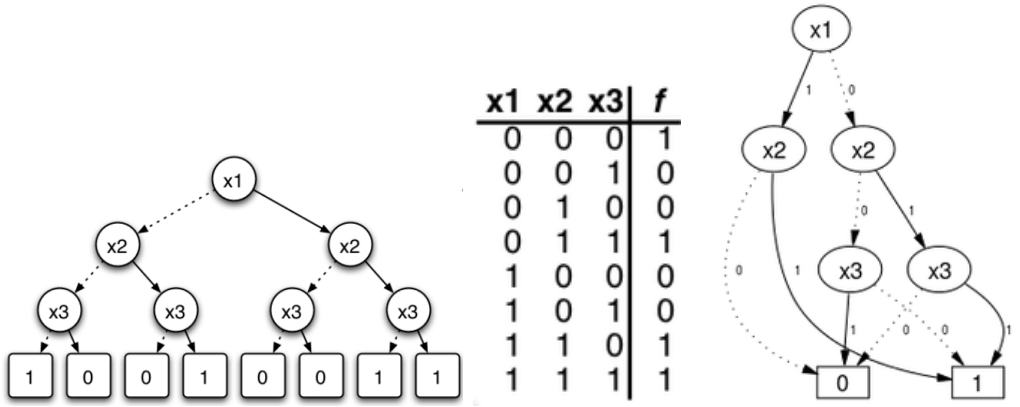
1.2 Traditional Boolean Verification Methods

1.2.1 Binary Decision Diagram (BDD)

Binary Decision Diagrams (BDDs) [14] and their variants, such as BMDs [13], TEDs [19] and FDDs [8], belong to the class of Canonical Diagrams. Of particular importance is BDD, an efficient data structure to represent and manipulate Boolean functions. BDDs are minimal, canonical and irreducible representation derived from *Shannon expansion* [37].

BDD is a rooted, directed, acyclic graph, whose nodes represent binary decisions. Each node represents a binary variable v and has two children: one (positive cofactor) representing the function with variable taking value $v = 1$, the other one (positive cofactor) with $v = 0$. There are also two terminal nodes, constant 1 and constant 0. The Boolean function encoded in a BDD is obtained by enumerating all the paths from the root to constant node 1. An example of a BDD is shown in Fig. 1.2(c) for Boolean function $f(x_1, x_2, x_3) = \bar{x_1}\bar{x_2}\bar{x_3} + x_1x_2 + x_2x_3$. For comparison, its function is also shown as a truth table Fig. 1.2(b).

The BDDs are typically ordered such that along any path from root to constant node 1, the variables appear in the same order. By construction, a BDD is minimal and irreducible, in the sense that there are no two nodes that correspond to the



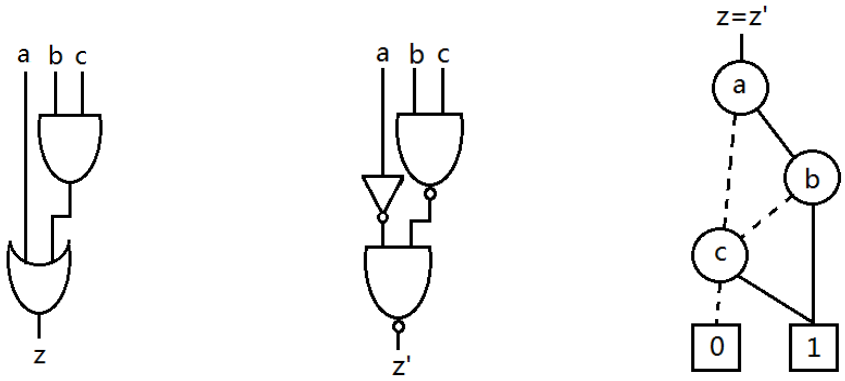
a) Decision Tree b) Truth Table c) Binary Decision Diagram

Figure 1.2: Different representations of Boolean Function

same function. Such BDDs are called *Reduced Ordered Binary Decision Diagrams* (ROBDDs) [70], referred to as BDDs for short. An important feature of the BDD is that it is canonical. This feature makes it possible to check equivalence between two functions. This is done by constructing their BDDs for the same order of variables and checking if they are isomorphic.

The example of equivalence checking using BDD is shown in Figure1.3. The Boolean function has been mapped into two different logic designs z and z' as shown in Figure1.3 (a) and Figure1.3 (b). Since BDD is canonical for a given variable order, EC can be done by comparing the BDDs of these two designs with the same variable ordering. In this example, the variable is orders in $a \rightarrow b \rightarrow c$. Since their BDDs are identical (see Figure 1.3 (c)), these two designs are functionally equivalent.

However, the BDD-based equivalence checking has its limitations since the size of the BDD grows dramatically for large designs, making it impractical for the verification of arithmetic circuits, especially the multipliers. For example, the BDD of a 4-bit integer multiplication has $1,022$ nodes, and for a 6-bit multiplication, the number of BDD nodes goes up to $8,176$. In general, for complex arithmetic circuits,



a) design 1 b) design 2 c) ROBDD of both designs

Figure 1.3: Equivalence checking using BDD.

the construction and composition of BDDs is computationally expensive, and the exponentially increasing number of BDD nodes can cause memory problem [22][46].

1.2.2 Boolean satisfiability problem (SAT)

To mitigate the limitations of BDDs, other techniques have been developed to reduce the complexity of equivalence checking and other verification tusks. One of them that made a significant impact on the verification field is Boolean Satisfiability (SAT). The goal of SAT is to find an assignment of variables for which a given Boolean formula evaluates to 1. Typically the formula is given in a *conjunctive normal form* (CNF) [81], a conjunction of one or more clauses, where a clause is a disjunction of literals. For example, Boolean formula $\varphi = (a + \neg b)(\neg a + \neg b + c)$ can be satisfied by choosing $\{\ a = 1; b = 0; c = 0\ \}$, which makes $\varphi = 1$. If the assignment of variables that makes the Boolean formula $\varphi = 1$ does not exist, the problem is called *unsatisfied* (unSAT).

Several SAT solvers have been developed to solve Boolean decision problems, such as GRASP [47], Chaff [52], Lingeling [6] and MiniSAT [72]. All SAT solvers are based on the basic *Davis-Putnam- Logemann-Loveland* (DPLL) [26] algorithm, a backtracking-based search algorithm, introduced originally by Davis and Putnam in

1969 [83]. Many newer techniques, such as non-chronological backtracking, resolution, recursive learning, etc., have been developed to improve the efficiency of SAT solver. Modern SAT solvers come in two flavors: "conflict-driven" and "look-ahead". Conflict-driven solvers, such as *MiniSAT* [72], augment the basic DPLL search algorithm with efficient conflict analysis. The Conflict-Driven Clause Learning (CDCL) provides ability to learn new clauses that prevents the space search from ending in an unsatisfying assignment. Look-ahead solvers, such as *march_dl* [18], have strengthened reductions and improved the heuristics.

SAT methods are widely used in formal verification. In particular, they are used to check equivalence between two circuits by creating a so-called *miter* and proving that its output is unSAT. An example of a miter configuration is shown in Fig. 1.4, where two circuits F, G are connected by a cluster of XORs and an OR gate. Typically, one circuit is considered to be the reference (golden) circuit, known to be correct, and the other one is the implementation that one wants to verify.

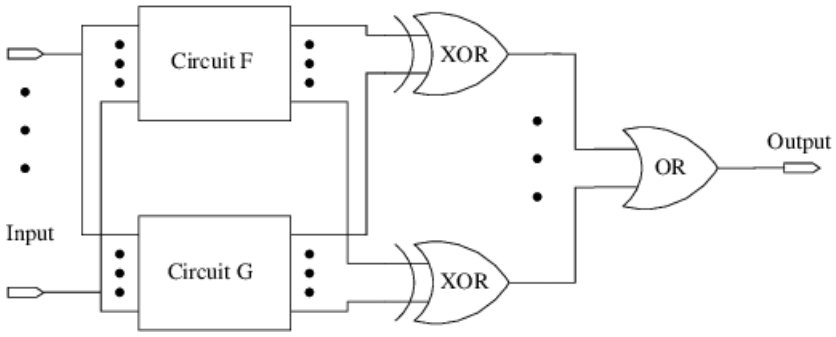


Figure 1.4: Using miter to solve equivalent checking in SAT

If the two circuits are functionally equivalent, then, for any input assignment, the same values should be observed at their outputs. In this case, none of the XORs would produce output 1, and the output of the OR gate should be 0. On the other hand, if different output values are computed by the two circuits for some input assignment, the output of the OR gate will be 1. Using this argument, the equivalence checking

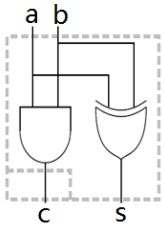
problem can be solved by checking if the output of the miter is *unSAT* (i.e., it never evaluates to 1). The expression of the miter's output is constructed from the circuits components in their CNF form. If the output of the miter is unSAT for all possible input assignment, the two circuits are equivalent. Otherwise, the problem is satisfiable and the SAT solver returns a *counterexample*, an assignment of inputs for which the two circuits produce different outputs.

The SAT-based method is particularly efficient at finding a satisfying solution. However, most of the circuit verification problems rely on checking if the problem is unSAT, as discussed above. Theoretically, in order to prove that the assignment of variables that evaluates a given Boolean formula to 1 does not exist, one has to evaluate all possible input assignment. Although there are many advanced algorithms that help refine the search space, the SAT method has a low scalability for large arithmetic circuits. For example, the state-of-the-art SAT solver, *miniSAT_blbd* [18], takes up to an hour to verify a 16-bit multiplier [74], and is ineffective in handling 32-bit and 64-bit wide multiplication in core datapaths.

1.2.3 Satisfiability Modulo Theories (SMT)

An extension of Boolean SAT is *Satisfiability Modulo Theories* (SMT) [59], which builds on SAT by incorporating different supporting theories. It is intended to solve more complex problem. Instead of treating the problem in a strictly Boolean domain, the SMT solvers integrate different well-defined theories (Boolean logic, bit vectors, integer and real arithmetic, linear inequalities, uninterpreted functions, arrays, lists, etc.) into a DPLL-style SAT decision procedure. By doing so, SMT solvers can solve a satisfiability problem of a word-level miter. For example, to check if the circuit shown in Figure 1.5(a) implements a Half-adder (HA), a word-level miter $z = a + b - (2c + s)$ is generated. The CNF formula of the miter is shown in 1.5(b). The last equation

models the miter as bit-vector adding $(a+b)$ and $-(2c+s)$, where the constant "10" represents bit-vector of the integer constant 2.



a) Half adder with input a, input b, output s as the sum and output c as the carry.

$$(\bar{a}+\bar{b}+\bar{s})(a+b+\bar{s})(a+\bar{b}+s)(\bar{a}+b+s)$$
$$\wedge\,(a+b+\bar{c})(\bar{a}+c)(\bar{b}+c)$$
$$\wedge\,(bvadd\ (bvadd(a,b)\ ,\ \neg bvadd(bvmul(10,c),s)))$$

b) CNF formula of word-level miter.

Figure 1.5: Using SMT method to solve world-level miter.

Some of the modern commonly known SMT solvers include Boolector [55], Z3 [28] and CVC [73]. However, SMT solvers still model the problem as a decision problem, and are not efficient at verifying large arithmetic circuits as demonstrated in the literature and confirmed in our experiments.

1.2.4 Theorem Proving

Another class of verification solvers, particularly popular in industry, are Theorem Provers, which based on a deductive proof system. The proof system is usually based on a strongly problem-specific database of axioms and inference rules, such as simplification, rewriting, and induction. The most common theorem proving systems are: HOL [36], PVS [56], Boyer-Moore/ACL2 [12], and Nqthm [41]. These systems are characterized by high abstraction and powerful logic expressiveness. The use of a general mathematical framework offers some advantages that can be significant, or even essential, for some verification tasks.

However these systems are highly interactive, requiring user guidance and deep understanding of the design and the system. In order to prove that an implementation satisfies the specification, one has to describe their relation as theorems within the

context of a proof calculus. Building such a system from the scratch is difficult, since modeling of the gate-level circuits in theorem proving is extremely complex. Typically, using such systems is more difficult and time-consuming than using highly automated, methods like SAT or model checking [85][45]. Furthermore, the success of verification using theorem proving depends on the set of available axioms and rewrite rules, and on the choice their order, so it cannot always guarantee a conclusive answer [78][40]. With so many limitations, Theorem Proving is not a good choice for a general circuit verification. In the next two chapters, a more effective method is proposed based on computer algebra, which forms the foundation of this thesis.

1.3 Motivation

Traditional Boolean verification methods discussed in the previous section are unsuitable for verifying large arithmetic circuits. As mentioned earlier, the need for "bit blasting", i.e., flattening the design into bit-level netlist, makes their computation very expensive when the width of the data path is large. To address this issue, the method proposed in this work models the verification problem in an algebraic domain, rather than strictly Boolean. An efficient approach, called *algebraic rewriting*, based on Symbolic Computer Algebra is introduced in Chapter 3. While the basic rewriting approach has been proposed in our earlier work [86][21][22], in this thesis it is further refined and formalized.

Algebraic rewriting provides impressive results in verifying standard arithmetic circuit [86], such as adders and multipliers. However, when dealing with arithmetic circuits derived from the standard class, it becomes prohibitive (in terms of memory usage) as shown in the later experiments. Two classes for non-standard arithmetic circuits are discussed in this thesis: 1) *truncated circuits*, in which some of the output bits are truncated; 2) arithmetic circuits are subjected to some Boolean or arithmetic constraints. Little attention to the verification of such circuits has been given in the

literature so far. In Chapter 4 and Chapter 5, these problems are formally analyzed and efficient solutions are proposed.

The proposed computer algebra-based verification method can be applied not only to check if the circuit is functionally correct, but also to identify and correct the logic bugs. Bug identification and correction (debugging) are known hard problems. In general, logic debugging methods are heuristic, and their performance, and even success, strongly depends on the location of the bug in the circuit. In Chapter 6, we propose a debugging technique for which this problem can be solved efficiently on a class of arithmetic circuits, namely Galois Field multipliers. To the best of our knowledge, it is the first debugging method whose performance is not significantly affected by the bug location.

CHAPTER 2

BACKGROUND

This chapter provides mathematical background of computer algebra method used in this thesis and reviews the related work in the literature. Specifically, in order to build an algebraic model for an arithmetic circuit in the context of computer algebra, the following concepts are needed: fields, polynomials, ideals, varieties and ideal membership, and Gröbner basis.

2.1 Fields, Polynomials, Ideals and Varieties

2.1.1 Fields

In mathematics, a field is a set F, containing at least two elements, on which two operations $+$ and $\cdot$ (called addition and multiplication, respectively) are defined so that for each pair of elements x, y in F there are unique elements $x + y$ and $x \cdot y$ in F. A field is thus a fundamental algebraic structure, which is widely used in algebra, number theory, and many other areas of mathematics. To learn about fields, we start with the *commutative ring*, since field is a special class of ring. A commutative ring consists of a set R and two binary operations "$\cdot$" and "$+$" defined on R, for which the following conditions are satisfied:

(i) *Associativity*: $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c, \in R$.

(ii) *Commutativity*: $a + b = b + a$ and $a \cdot b = b \cdot a$ for all $a, b \in R$.

(iii) *Distributivity*: $a \cdot (b + c) = a \cdot b + a \cdot c$ for all $a, b, c \in R$.

(iv) *Identities*: There are $0, 1 \in R$ such that $a + 0 = a$ and $a \cdot 1 = a$ for all $a \in R$.

(v) *Additive inverses*: Given $a \in R$, there is $b \in R$ such that $a + b = 0$.

Two examples of commutative rings are the integers $\mathbb{Z}$ and the polynomial ring $k[x_1,...,x_n]$, with coefficients in an arbitrary field k. A *field* $\mathbb{F}$ is a commutative ring with unity, where every element in $\mathbb{F}$, except 0, has a *multiplicative inverse*: $\forall a \in (\mathbb{F} - \{0\}), \exists \hat{a} \in \mathbb{F}$ such that $a \cdot \hat{a} = 1$. The most commonly used fields are $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$. The set $\mathbb{Z}$, which is of particular interest to us, is a ring but not a field, since it does not have the attribution of multiplicative inverse.

2.1.2 Polynomials

A polynomial is an expression consisting of variables and coefficients, that involves the operations of addition, multiplication, and non-negative integer exponents of variables. In general, a **polynomial** f in variables $x_1,...,x_n$ is a finite linear combination of monomials, with coefficients in some field k. A polynomial can always be written in a sum-of-product form $f = \sum a_i x_i^{\alpha_i}$, where each product $x_i^{\alpha_i}$ is called *monomial* and a_i is the coefficient. A monomial in variables $x_1,...,x_n$ is a product of the form $x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdots x_n^{\alpha_n}$, where all of the exponents $\alpha_1,...,\alpha_n$ are nonnegative integers. The *degree* of this monomial is the sum $\alpha_1 + \cdots + \alpha_n$. The total degree of polynomial f, denoted $\deg(f)$, is the maximum degree among all the monomials. A *term* of f is the product of a nonzero coefficient and its monomial. As an example, polynomial $f = 2x^3y^2z + \frac{2}{3}y^3z^3 - 3xyz + y^2$ has four terms and total degree six. Note that there are two terms of maximal total degree, which is something that cannot happen for polynomials in one variable.

There are several ways to order monomials (referred to as *term order*), such as *lexicographic order* (LEX), *Degree reverse lexicographic order* (DEGREVLEX), and others. For instance, in LEX order, $2x^3y^2z > \frac{2}{3}y^3z^3$. The first, or greatest term of f (in terms of the adapted term order), is called the *leading term* $lt(f)$ of the polynomial f. In the above example, the leading term is $2x^3y^2z$.

Leading terms play an important role in the proposed verification method, where logic gates of a circuit are described as polynomials. Specifically, the polynomial terms are ordered such that the leading term represents a variable representing an output of a gate. This ordering makes a profound impact on the efficiency of the proposed verification technique. This issue will be discussed in detail in Chapter 3.

In this work, since all variables representing in the circuits are Boolean, we are particularly interested in polynomials with variables of degree 1. Such a polynomial is called *Pseudo-Boolean* polynomial. Formally, a Pseudo-Boolean function is a function $f : B^n \to \mathbb{R}$, where $B = \{0, 1\}$ is a Boolean domain and n is a nonnegative integer called the arity of the function. It can be written as a multi-linear polynomial

$$f = a + \sum a_i x_i + \sum_{i<j} a_{ij} x_i x_j + \sum_{i<j<k} a_{ijk} x_i x_j x_k + \dots$$

with constant coefficients $a, a_i, ...$ in the given field.

2.1.3 Ideals and Varieties

Given a polynomial ring $R = k[x_1,...,x_n]$ with coefficients in some field k, a subset $I \subset R$ is an ***ideal*** if it satisfies:

(i) $0 \in I$. (ii) If $f, g \in I$, then $f + g \in I$. (iii) If $f \in I$ and $h \in R$, then $hf \in I$.

In general, if $I \in k[x_1,...,x_n]$ consists of all the linear combinations of a set of polynomials $\{f_1, ..., f_s\} \in k[x_1,...,x_n]$, then I is an ideal of the set $\{f_1, ..., f_s\}$, and the set of $\{f_i\}$ is called *generator* or *basis*.

$$J = \langle f_1, ..., f_s \rangle = h_1 f_1 + ... + h_s f_s : \ h_i \in R \tag{2.1}$$

We call $\langle f_1, ..., f_s \rangle$ the ideal generated by the basis $\{f_1, ..., f_s\}$.

Given an ideal $J = \langle f_1, ..., f_s \rangle$ generated by $f_1, ..., f_s, \in k[x_1, ..., x_d]$, the set of all solutions to: $f_1 = f_2 = \cdots = f_s = 0$ is called ***variety*** $V(f_1, ..., f_s)$ of J. While

an ideal may have different bases, the variety depends only on the ideal and not on the basis (generator). That is, different bases that produce the same ideal will have exactly the same variety. In Section 2.3, we will introduce an especially useful basis for our verification, called Gröbner basis.

Let $\{f_1, ..., f_s\}$ and $\{g_1, ..., g_t\}$ be the bases of the same ideal in $k[x_1,...,x_n]$, i.e. $\langle f_1, ..., f_s\rangle = \langle g_1, ..., g_t\rangle$; then $V(f_1, ..., f_s) = V(g_1, ..., g_t)$. In the next section, we will show how the concept of ideal and variety is applied to circuit verification.

2.2 Ideal Membership Test

The symbolic algebra theories about polynomial rings, ideals and varieties we use in this work are all defined over a field, typically $\mathbb{Q}$. However, as described next and fully developed in the next section, the polynomials introduced in our work represent logic gates and are defined over ring $\mathbb{Z}$. However, these polynomials have a special structure, namely their leading term $lt(f_i)$ that represents a variable associated with a logic gate g_i, has coefficient 1. Subsequently, the process of polynomial division, which is an essential element of the verification process (to be described in detail later), will never introduce any coefficient outside of $\mathbb{Z}$. Consequently, this allows us to treat the polynomials as if they were in $\mathbb{Q}$.

Let $B = \{f_1, ...f_s\}$, with $f_i \in \mathbb{Z}[X]$, be a set of polynomials representing the circuit elements and let the ideal $J = \langle f_1, ..., f_s\rangle$ be generated by basis $\{f_1, ..., f_s\}$. In our case, each generator is a polynomial model of a circuit module (logic gate), and the set of generators can be viewed as the *implementation* of the circuit. Then, from the circuit perspective, the variety $V(J)$ of J, which is the set of all simultaneous solutions to a system of equations $f_1(x_1, ..., x_n) = 0; ..., f_s(x_1, ..., x_n) = 0$, contains all signal values of the circuit for all possible input valuations $\{x_i\}$.

Similarly, functional *specification* of the circuit is also defined as a polynomial in $\mathbb{Z}[X]$, where X is a set of inputs and outputs. For example, the specification of a

multiplier circuit, $Z = A \cdot B$, can then be written as a polynomial $F : Z - A \cdot B$. Here, A, B, and Z are symbolic, word-level variables, each represented as a polynomial in their respective bit-level variables, e.g., $A = \sum_{i=0}^{n-1} 2^i a_i$, and similarly for B and R. In terms of *computer algebra*, the arithmetic circuit verification problem is then formulated as follows [59][46][68][66]:

Given a circuit represented by a set of generators (implementation), $B = \{f_1, ..., f_s\}$, and the specification F, the goal of functional verification is to prove that the implementation (B) satisfies the specification (F). Here, B have the same notation as the previous example, but it represents the set of gate polynomials.

This means that for a functionally correct circuit, the solution to $F = 0$ agrees with $V(J)$, or, equivalently, that F vanishes on $V(J)$[1]. Consequently, this problem has been termed as an *ideal membership test*, which decides whether the specification polynomial F is a member of the ideal J generated by B, i.e., if $F \in J$ [35][59][46].

Given an ideal $J = \langle f_1, ..., f_s \rangle$, in order to test if $F \in J$, polynomial F is divided consecutively by $f_1, ..., f_s$. The goal of each division is to cancel the leading term of F (with respect to a chosen term order) using one of the leading terms of $f_1, ..., f_s$. Such a reduction results in a polynomial remainder $r = F - \frac{lt(F)}{lt(f_i)} \cdot f_i$, in which the leading term $lt(F)$ has been canceled. If the remainder r reduces to zero, the implementation satisfies the specification. However, if $r \neq 0$, such a conclusion cannot be drawn: r can still be in J but it is not divisible by any of the polynomials in $B = \{f_1, ..., f_s\}$. That is, the basis $B = \{f_1, ..., f_s\}$ may not be sufficient to reduce $F \longrightarrow 0$, and yet the circuit may be correct. To check if F is reducible to zero for the given ideal J, one must compute a *canonical* set of generators, $G = \{p_1, ..., p_t\}$, called the *Gröbner basis*, with the same ideal $\langle p_1, ..., p_t \rangle = \langle f_1, ..., f_s \rangle$. Let the set $G = \{p_1, ..., p_t\}$ be the Gröbner basis for ideal J, then F belongs to J if and only if the remainder of the

[1]Polynomial f is said to vanish on a set V if $\forall a \in V \ f(a) = 0$. Or, $V(f) \subseteq V(J)$.

division of F by the elements of G is zero, denoted as $\forall F \in J$, $F \xrightarrow{G}_+ 0$ [3]. The sign $+$ means that the division/reduction is done consecutively by using the elements of G one by one. In short, the Gröbner basis is necessary to unequivocally answer the question whether $F \in J$.

2.3 Gröbner basis

A basis $\{p_1, ..., p_t\}$ of an ideal $J\langle p_1, ..., p_t\rangle$ is called a ***Gröbner basis*** (w.r.t. the monomial order $>$) if the leading term of every nonzero element of J is a multiple of (at least) one of the leading term $lt(p_1), ..., lt(p_t)$. A known algorithmic procedure for computing a Gröbner basis is called Buchberger's algorithm [15]. Given some basis $B = \{f_1, ..., f_s\}$, it produces another basis $G = \{p_1, ..., p_t\}$, such that the ideals $\langle p_1, ..., p_t\rangle = \langle f_1, ..., f_t\rangle$ and hence have the same variety $V(\langle G\rangle) = V(\langle B\rangle)$. Buchberger's algorithm is computationally expensive, since it computes the so-called *S-polynomials (SPoly)* by performing reduction operations on all pairs of polynomials in B. The S-polynomial of polynomials p and g in a polynomial set P, is the combination $\text{Spoly}(p, g) = \frac{L}{lt(p)}p - \frac{L}{lt(g)}g$, where L is the least common multiple $\text{LCM}(lm(p), lm(g))$. Note that $\text{Spoly}(p, g)$ cancels the leading terms of p and g, and the remainder r obtained in $\text{Spoly}(p, g)$ $F \xrightarrow{P}_+ r$ gives a new leading term.

The basic purpose of computing SPoly pairs is to compute polynomials with new leading terms, which can be used in the reduction step of the ideal-membership testing. These newly generated polynomials belong to the ideal G which completely defines the system. To compute Gröbner basis $G = \{g_1, ..., g_l\}$ for an ideal $\langle p_1, ..., p_t\rangle$, Buchberger's algorithm computes G in some finite number of steps by performing the $\text{Spoly}(p, g) \xrightarrow{P}_+ r$ iteratively. The algorithm determines if $\text{Spoly}(p, g) \xrightarrow{P}_+ 0$. In this case, we also conclude that all polynomials are relatively prime to each other, with a distinct leading term.

This establishes that the generating set (generator) whose polynomials are relatively prime to each other is in fact a *Gröbner* basis. This important fact will be used in developing the verification method in the upcoming sections. A number of other algorithms have been developed for computing a *Gröbner* basis, such as $F4$ [34], which in contrast to the basic Buchberger's algorithm, compute multiple SPoly pairs in each iteration. However, in general, the process of generating a *Gröbner* remains computationally expensive.

2.4 Related Work

The work in arithmetic circuit verification was pioneered by Shekhar et al. [69] and Wienand et al. [82], where some important concepts from computer algebra and algebraic geometry were applied to model the core verification problem. In [82] an arithmetic circuit is modeled as a network of arithmetic operators, such as half- and full-adders, comparators, and product generators, extracted from the gate-level implementation. These operators are modeled using *arithmetic bit-level* (ABL) expressions, $B = \{B_j\}$. The authors of [82] (and also of [46]) show that for an arbitrary combinational circuit, if the terms of the gate equations B are ordered in reverse topological order, $\{\text{outputs}\} > \{\text{inputs}\}$, then all leading monomials of the polynomials in B are relatively prime. As a result, the corresponding set B already constitutes a Gröbner basis (GB), obviating the computation of the complete canonical Gröbner basis. The verification problem is solved by reducing the specification F modulo B to a *normal form* and testing if it vanishes over $\mathbb{Z}_{2^n}$. The restriction to binary variables is achieved by imposing Boolean constraints, $\langle x^2 - x \rangle$ for all the variable x [59], and the problem is solved over quotient ring $Q = \mathbb{Z}_{2^n}[X]/\langle x^2 - x \rangle$ (for all variable x) using a popular computer algebra system, Singular [29]. This approach, however, is limited to circuits composed entirely of half adders and full adders, which must first be extracted from the gate-level implementation. In practice, this is the most expensive part of the

process, and it is not always possible to perform such extraction, especially in highly bit-optimized implementations.

In [46] the verification problem was similarly formulated as an *ideal membership test* but applied to Galois Field (GF or $\mathbb{F}_{2^q}$) arithmetic circuits. It has been shown that in GF, when the specification F and the ideal J of the circuit implementation are in $\mathbb{F}_{2^q}$, the problem can be reduced to testing if $F \in (J + J_0)$, over a larger ideal $(J + J_0)$ where $J_0 = \langle x^2 - x \rangle$ is an ideal of the field polynomials. Adding J_0 basically restricts the variety V to solutions in $\mathbb{F}_2$, i.e. to $V(J) \cap V(J_0)$ [25]. The polynomials of J_0 are referred to as *field polynomials*. Similarly to [82], the authors of [46] derive the term order from the topological structure of the circuit, which renders the set of polynomials B (circuit implementation) a Gröbner basis (GB), thus obviating the need to perform the expensive GB computation. The method uses a customized, F4-style polynomial reduction using a modified Gaussian elimination algorithm [34] under this term order.

A different approach has been proposed in [86], whereby the expensive polynomial reduction has been replaced by a computationally simpler *algebraic rewriting* technique. The method introduces the concept of an *input signature*, a polynomial in the primary inputs, and an *output signature*, a polynomial derived from the encoding of the primary outputs. The verification is accomplished by rewriting the output signature, using algebraic expressions of the internal gates, into an input signature. This process de facto performs *function extraction*. Several ordering techniques have been described to make this method applicable to large arithmetic circuits, but the method still cannot handle the heavily optimized circuits.

A similar approach to arithmetic circuit verification, called *backward construction*, was proposed in 1995 in [37]. It uses BMDs to reconstruct functional, high level representation from the gate-level structure of arithmetic circuits such as adders and multipliers. Experimental results show that time complexity of the tested circuits is

in the order of n^4 for multipliers with n bit operands. There is no clear indication if the BMD is an efficient data structure for this problem.

The basic approach of the ideal membership testing and Gröbner basis (GB) reduction has also been used in the works of [68][66], where it was applied to integer circuits. In [68] the following features have been added to make the reduction more efficient: 1) *Logic reduction* with an AND-XOR vanishing rule, which analyzes the structure of the circuit to identify and remove vanishing monomials that correspond to the product of XOR, AND signals with shared input variables; 2) An *XOR rewriting scheme*, which reduces the model of the circuit to consider only primary inputs, outputs, and fan-out points/XOR gates; and 3) *Common rewriting*, which eliminates the nodes with a single parent. These techniques simplify the task of GB reduction by eliminating all the nodes which have exactly one parent, thus increasing the chance for early term cancellation during the rewriting process.

Another work [66] revisits the techniques from [86] and [68] and provides the proof of correctness for these approaches. It uses a column-wise technique to model and verify basic multiplier structures by computing the Gröbner basis incrementally for each column of the output bit, rather than for the entire circuit. The paper justifies the use of the theory of ideal membership (in principle applicable to $\mathbb{Q}[X]$) to prove properties of integer arithmetic circuits in $\mathbb{Z}$. It points out that, since the leading coefficients of the gate polynomials forming the Gröbner basis are +1 or -1, polynomial reduction never introduces fractional coefficients and their computation remains in $\mathbb{Z}$. This also explains why the dedicated implementations in [86] and [68] can rely on computation in $\mathbb{Z}$ only, while remaining sound and complete [66]. A follow-up paper [65] describes an enhancement to this column-wise technique by extracting half- and full-adder constraints to further reduce the size of Gröbner basis to speed up the reduction process.

In general, the problem of formally verifying complex integer arithmetic circuits (not just multipliers) remains open, and new solutions are being proposed. In the next chapter, an efficient and scalable approach, called *algebraic rewriting*, is formally introduced to address this issue. This approach has already been proposed by our group earlier, but it is further refined and formalized in this thesis. In addition, a *bit-flow* model is proposed to support the proof of the correctness of algebraic rewriting, and to offer a new insight into the problem of arithmetic circuit verification [20].

CHAPTER 3

FORMAL VERIFICATION OF INTEGER ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA APPROACH

This chapter introduces an algebraic model used in circuit verification, which is the key to solve the verification problem in algebraic domain. Two flavors of computer algebra techniques that use this model will be discussed in detail: 1) Gröbner basis reduction techniques [59][68][66] and 2) algebraic rewriting [86]. Detailed algorithms for the reduction and the rewriting are given. We analyze the relation between these two computer algebra techniques and provide a comparison from the efficiency point of view.

3.1 Algebraic Model of Electronic Circuit

The arithmetic circuits considered in this thesis are circuits whose computation can be expressed as a polynomial in the input variables. These include adders, subtractors, multipliers, fused add-multiply circuits, dividers, etc.. The circuit is modeled as a network of interconnected bit-level components, each with a finite set of binary inputs and one or more binary outputs. In this work we will focus on *gate-level* integer arithmetic circuits with single-output logic gates. However, the model can be extended to other, more complex and multiple-output circuit components.

Each gate is modeled by a *pseudo-Boolean* polynomial $f_i \in \mathbb{Z}[X]$, with Boolean variables X representing circuit signals associated with the gate. It is an algebraic expression with usual multiplication and addition operators over Boolean variables. Formally, a pseudo-Boolean polynomial is an integer-valued function $f : \{0,1\}^n \to \mathbb{Z}$.

The following expressions summarize the algebraic representation of basic Boolean operators NOT, AND, OR and XOR.

$$\begin{aligned} \neg a &= 1 - a \\ a \wedge b &= a \cdot b \\ a \vee b &= a + b - a \cdot b \\ a \oplus b &= a + b - 2a \cdot b \end{aligned} \tag{3.1}$$

By construction, each expression evaluates to a binary value {0,1} and hence correctly models the Boolean function of a logic gate. Models for more complex AOI (And-Or-Invert) gates, used in standard cell technology, are readily obtained from these basic logic expressions. For example, the algebraic model for the logic gate $g = a \vee (b \wedge c)$ can be derived as $g = a + bc - abc$, etc. Similarly, a 3-input OR gate can be represented as $z = a+b+c-ab-ac-bc+abc$, a 3-input XOR gate as $z = a+b+c-2ab-2ac-2bc+4abc$, etc.

Multiple output modules, such as single-bit adders, with binary inputs can be expressed similarly. For example, a half-adder (HA) and a full-adder (FA), can be expressed by the following expressions:

$$\begin{aligned} \text{HA}: &\quad 2C + S = a + b \\ \text{FA}: &\quad 2C + S = a + b + c_{in} \end{aligned} \tag{3.2}$$

where a, b, c_{in} are binary inputs and C, S are binary outputs.

The function computed by an arithmetic circuit is represented as a *specification* polynomial in the primary input variables, denoted F_{spec}. For example, the specification of an n-bit unsigned integer multiplier, $Z = A \cdot B$ with inputs $A = [a_0, \cdots, a_{n-1}]$ and $B = [b_0, \cdots, b_{n-1}]$, is described by $F_{spec} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} a_i b_j$. The result of the computation, stored in the primary output bits, is also expressed as a polynomial, called *output signature*, S_{out}. Typically, such a polynomial is linear, uniquely

determined by the m-bit encoding of the output, provided by the designer. For example, for a signed 2's complement arithmetic circuit with m output bits, $S_{out} = -2^{m-1}z_{m-1} + \sum_{i=0}^{m-2} 2^i z_i$. The circuit is implemented as a network of logic gates G, each modeled as a polynomial g_i derived from Eqn.(6.1). The polynomial representing a given gate evaluates to zero for all the input and output combinations satisfied by this gate. As an example, a non-standard gate-level implementation of a full adder, is shown in Fig. 3.1.

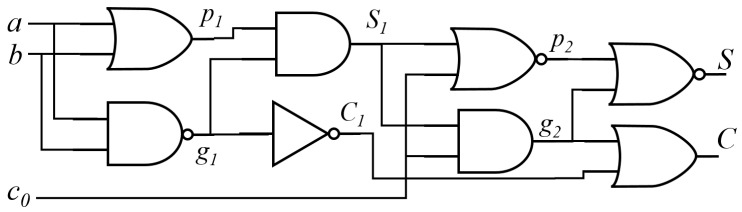


Figure 3.1: Gate-level arithmetic circuit (Full Adder)

The set of polynomials $G = \{f_i\}$ in Eqn. 3.3 represents the gate-level implementation of the full adder circuit. We refer to this set as G to indicate that it is a Gröbner basis (or GB for short). It has been shown that if the polynomials in G are ordered such that the leading term is the output of the gate, and the leading term of all the polynomials are relatively prime, the set G forms Gröbner basis [62].

The set G consists of two parts: *Gate polynomials* $(f_1, ..., f_8)$ and *Field polynomials* $(f_9, ..., f_{17})$. Each gate polynomial satisfies the relation $f_i = 0$. The gate polynomials have the form $f_i = v_i - tail(f_i)$, where the leading term $lt(f_i) = v_i$ is the output of gate f_i, and $tail(f_i)$ is the logic specification of the gate in terms of its inputs. The leading terms under such ordering are relatively prime, which renders G a Gröbner basis [59][46][66]. This feature is essential for both the GB reduction and algebraic rewriting, which will be discussed in the next sections.

$$
\begin{aligned}
f_1 &= p_1 - (-ab + a + b) \\
f_2 &= g_1 - (-ab + 1) \\
f_3 &= S_1 - p_1 g_1 \\
f_4 &= C_1 - (-g_1 + 1) \\
f_5 &= p_2 - (S_1 c_0 - S_1 - c_0 + 1) \\
f_6 &= g_2 - S_1 c_0 \\
f_7 &= S - (p_2 g_2 - p_2 - g_2 + +1) \\
f_8 &= C - (-C_1 g_2 + C_1 + g_2) \\
f_9 &= (a^2 - a) \\
f_{10} &= (b^2 - b) \\
&\cdots\cdots \\
f_{17} &= (g_2^2 - g_2)
\end{aligned}
\tag{3.3}
$$

Each field polynomials, $f_9, ..., f_{17}$, has the form $J_0 =< x^2 - x >$, where x is one of the signals $\{a, b, c_0, p_1, g_1, S_1, C_1, p_2, g_2\}$. They play an important role in polynomial reduction to maintain the Boolean property of each variable. However, they are handled differently in the GB reduction than in the algebraic rewriting approach, as discussed in the next sections.

3.2 Gröbner Basis Polynomial Reduction

In this method the reduction of F modulo G is accomplished by successively eliminating terms of F, one by one, by a leading term of some polynomial $f_i \in G$, using Gaussian elimination. The reduction is performed over a Gröbner basis derived from G and the field polynomials J_0. From the mathematical point of view, this means that the computation will be performed in the quotient ring, $\mathbb{Z}[X]/\langle x^2 - x\rangle : x \in X$, the set of all variables (signals) of the circuit. The Gröbner basis (GB) reduction algorithm is given in Algorithm 1. First, the polynomial base G=$\{f_1, ..., f_m\}$ is derived from

$\mathcal{N}$ using Equations (6.1), where m is the number of logic components in $\mathcal{N}$. Each polynomial in G has the form $f_i = v + tail(f_i)$, where v is the the leading monomial $lm(f_i)$. All the variables in the circuit are ordered in reverse-topological order, from primary outputs to primary inputs, and for each gate polynomial from the gate output to its inputs.

Furthermore, the output signals of the gates that depend on common variables (fanins) should be ordered next to each other, as this will maximize the chance for a potential term cancellation and minimize the size of intermediate polynomials. For example, consider the reduction of a polynomial $F = 2C + S + ...$ in a circuit containing a half adder composed of an AND gate $C = ab$ and an XOR gate $S = a+b-2ab$. Since both C and S depend on common variables, a, b, reducing them one immediately after the other will eliminate the product term ab from the polynomial, resulting in $F = a+b+.....$ This is beneficial from the complexity point of view, and should be performed before the reduction of the remaining terms of the polynomial.

Considering these two basic ordering rules, one possible term order for the polynomial ring of the circuit in Figure 3.1 is shown below, where variables in curly brackets can assume any relative order.

$$\{S, C\} > \{p_2, g_2\} > \{S_1, C_1\} > \{p_1, g_1\} > \{a, b, c_0\} \tag{3.4}$$

The expression F to be reduced is initialized with the difference between the output signature S_{out} and F_{spec}. In this case $F = 2C + S - (a + b + c_0)$. The goal is to reduce F to 0 by G.

The main part of the GB reduction is given in lines 5-15. The algorithm searches for a polynomial f_i in G such that the leading term of f_i divides the current leading term $lt(F)$ of F. If such a polynomial exists, it will be used to reduce F, as shown in line 8. Otherwise, the $lt(F)$ will be moved to the remainder Rem (lines $11 - 12$). At any point, when new terms (containing new intermediate variables introduced by

Algorithm 1 Gröebner Basis Polynomial Reduction

Input: Specification polynomial F_{spec};and Gate-level netlist $\mathcal{N}$
Output: Remainder Rem

1: Create base G=$\{f_1,...,f_m\}$ of $\mathcal{N}$ using Eq.(6.1)
2: Generate S_{out} from $\mathcal{N}$
3: Define ring and specify term order
4: Initialize $F \leftarrow S_{out} - F_{spec}$
5: **while** $F \neq 0$ **do**
6: **if** $\exists f_i \in G : \frac{lt(F)}{lt(f_i)} \neq 0$ **then**
7: /* there exists f_i such that its leading term is divisible by $lt(F)$ */
8: $F \leftarrow F - \frac{lt(F)}{lt(f_i)} \cdot f_i$ // polynomial division
9: **else**
10: /* no leading term of f_i divides F, move $lt(F)$ to Rem */
11: $F \leftarrow F - lt(F)$
12: $Rem \leftarrow Rem + lt(F)$
13: **end if**
14: Maintain the term order imposed on the ring
15: **end while**
16: **return** Rem

division) are added to polynomial F (line 8), the procedure must maintain the term order imposed on the ring. The reduction process terminates when F becomes empty, either by being reduced or moved to Rem. The zero remainder is the evidence of a correct implementation, as discussed in Chapter 2.2.

We illustrate the GB reduction process with the example in Fig. 3.1. The initial polynomial for this circuit is:

$$F = 2C + S - (a + b + c_0) \tag{3.5}$$

Equation (3.6) gives the sequence of steps that reduces F with the gate polynomials $f_i \in G$ for the circuit in Figure 3.1. At each step, F represents the polynomial reduced by the previous reduction step. For brevity, the substitution is shown for a pair of variables at once. For example, $F/(C, S)$ means reducing variables C and S with polynomial f_8 followed by f_7. The term order given in Eqn. (3.4), imposed on the ring, is maintained throughout the entire reduction process.

$$
\begin{aligned}
F &= 2C + S - (a + b + c_0) \\
1)\ F/(\ S\ ,\ C\ ) &= 2(-C_1 g_2 + g_2 + C_1) + (p_2 g_2 - p_2 - g_2 + 1) - (a + b + c_0) \\
&= p_2 g_2 - p_2 - 2 g_2 C_1 + g_2 + 2C_1 - (a + b + c_0) + 1 \\
2)\ F/(\ p_2,\ g_2\ ) &= (S_1 c_0 - S_1 - c_0 + 1) S_1 c_0 - (S_1 c_0 - S_1 - c_0 + 1) - 2 S_1 C_1 c_0 \\
&\quad + S_1 c_0 + 2C_1 - (a + b + c_0) + 1 \\
&= \mathbf{S_1^2 c_0^2 - S_1^2 c_0 - S_1 c_0^2 + S_1 c_0} - 2 S_1 C_1 c_0 + S_1 + 2C_1 - (a + b) \\
3)\ F/(S_1^2 - S_1) &= -2 S_1 C_1 c_0 + S_1 + 2C_1 - (a + b) \\
4)\ F/(\ S_1,\ C_1) &= -2(p_1 g_1)(-g_1 + 1) c_0 + p_1 g_1 + 2(-g_1 + 1) - (a + b) \\
&= -2(\mathbf{-p_1 g_1^2 + p_1 g_1}) \mathbf{c_0} + p_1 g_1 - 2 g_1 - (a + b) + 2 \\
5)\ F/(\ g_1^2 - g_1) &= p_1 g_1 - 2 g_1 - (a + b) + 2 \\
6)\ F/(\ p_1,\ g_1\ ) &= (-ab + a + b)(-ab + 1) - 2(-ab + 1) - (a + b) + 2 \\
&= \mathbf{a^2 b^2 - a^2 b - a b^2 + ab} \\
7)\ F/(\ a^2 -\ a) &= 0
\end{aligned}
\tag{3.6}
$$

The effect of field polynomials $J_0 =< x^2 - x >$, responsible for keeping each variable Boolean, can be observed during steps 2, 4, 6 and 7, shown in bold. The reduction terminates in $Rem = 0$, indicating that the circuit implements the function indicated by the specification, a full adder.

3.3 Algebraic Rewriting

Algebraic rewriting is the process of transforming the output signature S_{out} into an input signature S_{in} using algebraic models of the internal components (logic gates) of the circuit. The rewriting is done in reverse topological order: from the primary outputs (PO) to the primary inputs (PI); for this reason it is also referred to as a *backward rewriting* [86]. Intermediate expressions obtained during rewriting are also represented as polynomials, referred to as *signatures*, over the variables representing the internal signals of the circuit. By construction, each variable in a given signature (starting with S_{out}) represents an output of some logic gate.

The rewriting transformation simply replaces each variable with the corresponding algebraic expression of the logic gate. If the variable is part of a monomial involving

other variables, the expression is multiplied by the remaining terms and expanded to a disjunctive normal form. This is followed by a standard polynomial simplification by combining terms with same monomials.

Algorithm 2 Algebraic Rewriting

Input: Specification polynomial F_{spec}; and Gate-level netlist $\mathcal{N}$
Output: $(S_{in} == F_{spec})$, or the computed signature S_{in}

1: Derive G=$\{f_1,...,f_m\}$ from $\mathcal{N}$ using Eqn.(6.1)
2: Sort G to maximize the cancellations // pre-processing
3: Generate S_{out} from $\mathcal{N}$
4: Initialize $Sig \leftarrow S_{out}$
5: **for** f_i in G **do**
6: $v \leftarrow \text{lm}(f_i)$ // leading monomial of f_i is output of a gate
7: **if** $v \in Sig$ **then**
8: /* replace v with $tail(f_i)$ in Sig */
9: $Sig \leftarrow Sig(v \leftarrow tail(f_i))$
10: $x \leftarrow x^2$ // for all x in Sig
11: **end if**
12: **end for**
13: /* upon termination, Sig is composed of PIs only */
14: **if** $Sig == F_{spec}$ **return True**
15: **else return** $S_{in} = Sig$

Algebraic Rewriting procedure is summarized in Algorithm 2. First, the polynomial base G=$\{f_1,...,f_m\}$ is derived from $\mathcal{N}$ using Eq.(6.1), as in the GB reduction. Then, the polynomials in G are sorted in reverse-topological order (lines 1-2). Among several possible topological orders the one that maximizes the number of early cancellations during rewriting is sought. This has an effect of minimizing the size of the intermediate polynomials during rewriting (the "fat belly" effect) [86]. It is accomplished by keeping together the polynomials whose leading terms (gate outputs) depend on common variables, as in the GB reduction. The expression to be rewritten, Sig, is initialized with the given output signature S_{out} of $\mathcal{N}$ (lines 3-4).

The main part of the rewriting, lines 5-12, iterates over the polynomials $f_i \in G$ and performs the required substitutions. Specifically, all occurrences of $v = lt(f_i)$ in Sig are replaced by $tail(f_i)$, followed by possible expansion of the resulting term.

To maintain Boolean values of the variables during rewriting, the degree of each variable in Sig is reduced to 1 (line 10). This step is equivalent to dividing Sig by a field polynomial $< x^2 - x >$, but it is achieved in a more efficient way. At the end, the algorithm returns S_{in} as the derived signature of the circuit. If the terms of polynomials in G are sorted in a reversed topological order, the returned polynomial S_{in} contains only the primary input (PI) variables, so it can be compared with F_{spec}.

While the main goal of algebraic rewriting, as described by Algorithm 2, is to determine the arithmetic function implemented by the circuit, it can also be used to verify it against the known specification. This can be simply done by rewriting $F = S_{out} - F_{spec}$ and checking if it produces a zero. We will use this rewriting mode in order to compare it against the GB reduction method in Chapter 3.2.

$$
\begin{aligned}
F &= 2C + S - (a + b + c_0) \\
1)\ F/(S,\ C\ ) &= 2(C_1 + g_2 - C_1 g_2) + (1 - (p_2 + g_2 - p_2 g_2)) - (a + b + c_0) \\
&= 2C_1 + g_2 - 2C_1 g_2 - p_2 + p_2 g_2 + 1 - (a + b + c_0) \\
2)\ F/(p_2,\ g_2) &= 2C_1 + S_1 c_0 - 2S_1 C_1 c_0 - (1 - (S_1 + c_0 - S_1 c_0)) \\
&\quad + (1 - (S_1 + c_0 - S_1 c_0))S_1 c_0 + 1 - (a + b + c_0) \\
&= 2C_1 - 2S_1 C_1 c_0 + S_1 + \mathbf{S_1 c_0} - \mathbf{S_1^2 c_0} - \mathbf{S_1 c_0^2} + \mathbf{S_1^2 c_0^2} - (a + b) \\
&= 2C_1 - 2S_1 C_1 + S_1 - (a + b) \\
3)\ F/(S_1, C_1) &= 2(1 - g_1) - 2(1 - g_1)(p_1 g_1)c_0 + p_1 g_1 - (a + b) \\
&= 2 - 2g_1 - 2(\mathbf{p_1 g_1} - \mathbf{p_1 g_1^2}) + p_1 g_1 - (a + b) \\
&= 2 - 2g_1 + p_1 g_1 - (a + b) \\
4)\ F/(p_1,\ g_1) &= 2 - 2(1 - ab) + (a + b - ab)(1 - ab) - (a + b) \\
&= \mathbf{ab - a^2 b - ab^2 + a^2 b^2}\ =\ 0
\end{aligned}
\tag{3.7}
$$

We illustrate the rewriting process using the example of the gate-level full-adder circuit in Figure 3.1. The output signature of the circuit is $S_{out} = 2C + S$, determined by the binary encoding of the output. The specification for this circuit $F_{spec} = a+b+c_0$. Following the ordering rules described in [86], the best rewriting order which minimizes the size of intermediate polynomials is $\{(S, C), (p_2, g_2),\ (S_1, C_1), (p_1, g_1)\}$, as in the GB reduction. The signals shown in brackets can be rewritten in any order

as they depend on common inputs. Equation (3.7) shows the rewriting steps for the circuit. The terms shown in bold face indicate those that are reduced to zero during polynomial simplification. For brevity, the substitution is shown for each pair of variables applied at once. For example: $F/(C, S)$ means rewriting of F using C and S variables of polynomials f_8, f_7.

During the rewriting, two types of simplifications can be observed:

- Simplification of the terms with same monomials; for example, $2g_2 - g_2 = g_2$, in Step 1. In the process, some polynomial terms are reduced to 0. This is a common simplification applied in GB reduction as well.

- Lowering the term x^2 to x, since the signal variables are binary. This can be seen in Steps 2, 3, and 4, shown in bold face. For example, in step 2 we have: $S_1c_0 - S_1^2c_0 - S_1c_0^2 + S_1^2c_0^2 = S_1c_0 - S_1c_0 - S_1c_0 + S_1c_0 = 0$. Similarly, in step 3: $(p_1g_1 - p_1g_1^2) = p_1g_1 - p_1g_1 = 0$, etc. This simplification is simpler and can be executed faster than dividing the polynomials by the respective field polynomials $(x^2 - x)$, as it is done in computer algebra approach. This is one of the main reasons for greater efficiency of the algebraic rewriting compared to GB reduction.

Subsequently, the final result reduces $F = S_{out} - F_{spec}$ to zero, indicating that the circuit correctly implements a full adder.

It should be noted that in addition to the two basic simplification rules mentioned above (rewriting the gates with common inputs, and the $x^2 \rightarrow x$ reduction), more sophisticated simplifications can be applied to the running polynomial Sig during rewriting by analyzing the structure of the gate-level network. For example, recognizing that some signal g is a product of XOR and AND signals with the same fanin inputs will reduce signal g to zero. This simplification, called an *XOR-AND vanishing*

rule has been used by [68], but for clarity of the above illustration, it has not been taken into account here.

3.4 AIG Rewriting

The algebraic rewriting technique described in the previous section can be further improved by performing rewriting using the functional AIG (Add-Inverter Graph) representation of the circuit instead of its gate level structure. This section provides a brief overview how this is accomplished, with details provided in [88].

AIG (And-Inverter Graph) is a combinational Boolean network composed of two-input AND gates and inverters [11]. Each internal node of the AIG represents a two-input AND function; the graph edges are labeled to indicate a possible inversion of the signal. We use the *cut-enumeration* approach of ABC [11] to detect XOR and Majority (MAJ) functions with a common set of variables; they are essential components of adder trees that are present in most arithmetic circuits in some form [88]. After detecting the XOR and MAJ components of the adder's AIG, rewriting skips over the detected adders, significantly speeding up the rewriting process. Figure 3.2 illustrates the process for the full adder (FA) circuit from Figure 3.1. In Figure 3.2 the groups of nodes (6,7,8) and (9,11,12) correspond to half adders (HA). The functions rooted at nodes 6 and 9 are majority (AND) functions, and those at nodes 12 and 8 are XORs. Subsequently, the functions at node 12 (S) and node 10 (C) are identified as XOR3 and MAJ3, respectively, on the shared inputs, a, b, c_0. The AIG rewriting of $S_{out} = 2C + S$ over the extracted XOR3 and MAJ3 nodes is trivial, with the nonlinear monomials automatically cancelled, as shown in Eqn. 3.8.

$$\begin{aligned} 2C + S &= 2(ab + ac_0 + bc_0 - 2abc_0) \\ &\quad + (a + b + c_o - 2ab - 2ac_0 - 2bc_0 + 4abc_0) \\ &= a + b + c_o \end{aligned} \tag{3.8}$$

The resulting signature matches the specification, which clearly indicates that the circuit is a full adder. As illustrated with this example, the AIG rewriting requires considerably fewer terms than the standard algebraic rewriting.

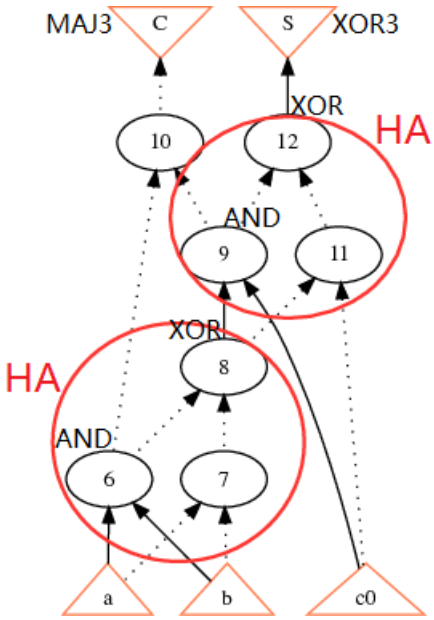


Figure 3.2: AIG rewriting of a full adder circuit from Figure 3.1.

Data structure: AIG rewriting is implemented in ABC with the polynomial data structure, type `Pln_Man_t`. Its main components include: 1) the AIG manager (`Gia_Man`) that represents the input design; and 2) two vector hash tables using type `Hsh_VecMan_t` are used for storing the constants and monomials. The hash tables of monomials include coefficient vectors and monomial vectors. When substitution is applied to the leading term, new monomials will be created and the substituted one will be removed. For example, when $ab + c + bd$ is substituted by $a = b + d$, the monomial ab is removed first, and b and bd are added to `Pln_Man_t`. During the process of adding the new monomials, the program will first check if these monomials already exist in `Pln_Man_t`; in this case only the coefficient of these monomials will be changed accordingly. In this example, two new monomials are generated by the

substitution, namely b^2, reduced to b, and bd. Since bd already exists in the expression, the coefficient 1 of bd is replaced by 2, resulting in $b + c + 2bd$.

3.5 Comparison between GB Reduction and Rewriting

It should be clear from the above discussion that both methods, the GB reduction and algebraic rewriting, are equivalent in the sense that they both perform polynomial reduction. The GB reduction scheme achieves polynomial reduction by division, in fact, performing Gaussian elimination. In contrast, algebraic rewriting does it by substituting the gate output variable by the polynomial expression of the gate's function. While the goal of GB reduction scheme is to reduce $F = S_{out} - F_{spec}$ modulo the set of implementation polynomials G to 0, it can also be used to extract the arithmetic function by reducing S_{out} modulo G, and interpret the result as the functional specification of the circuit F_{spec}. In the algebraic rewriting scheme, the goal is to rewrite the output signature S_{out} to S_{in}, the expression in the primary inputs, and check if it matches the expected specification F_{spec}. If $S_{in} = F_{spec}$, the circuit is correct; otherwise it is faulty. Alternatively, as illustrated above, algebraic rewriting can be also applied to $F = S_{out} - F_{spec}$, as in the GB approach.

Variable substitution of algebraic rewriting (line 9 of Algorithm 2) seems simpler than the main step of polynomial division of the GB reduction (line 8 of Algorithm 1). On the other hand, it requires additional multiplication of the terms and expansion into a sum of products. Hence, the complexity of these steps is comparable. Both methods avoid explicit computation of Gröbner basis, but achieve it by different means. In the GB reduction it is done by setting the variable order in the ring so that all variables are in reverse topological order, which makes the implementation set G a Gröbner basis. In the algebraic rewriting scheme on the other hand, the polynomials $f_i \in G$ are sorted in reverse topological order to effect the rewriting. As a result,

both methods ensure that the polynomial base is a Gröbner basis. However, there are some essential differences between the two methods that affect their efficiency.

- The GB reduction scheme requires the *field polynomials* $J_0 =< x^2 - x >$ to be added to the base G in order to keep the variables Boolean. This increases the size of the Gröbner basis and results in a larger search space in each iteration. Whereas in the rewriting scheme, the reduction by $< x^2 - x >$ is solved in a simpler way, namely by lowering x^2 to x via a simple data structure (line 10 in Algorithm 2).

- In the algebraic rewriting scheme, the gate polynomials $f_i \in G$ are ordered in reverse topological order (line 5 in Algorithm 2) so that each gate polynomial f_i is used exactly once. Furthermore, the selected polynomial is used to perform the rewriting by a simple string substitution and is never needed again. In contrast, in each iteration of the GB reduction one has to search for a polynomial f_i that divides the leading term of F under reduction. While in principle the GB reduction can also work over an ordered list of gate polynomials, this does not apply to the field polynomials $< x^2 - x >$, needed for the reduction. Since the appearance of intermediate signals in nonlinear terms x^k is unpredictable, it is impossible to pre-order the list of field polynomials in GB reduction.

Tables 3.1 and 3.2 show the verification results for multipliers mapped onto standard cells with three different libraries, including simple two-input gates and industrial libraries of 14 nm and 7 nm nodes. The table also compares the results with the open source tools of [66][65]. The first group of four designs in the table, labeled **-nomap*, are the circuits synthesized without technology mapping. The three circuits in the second group, labeled **-map-simple*, are synthesized and mapped onto a simple library of two-input gates. The last group of four circuits, labeled **-map-14nm*, contains designs that were synthesized and mapped onto a 14 nm industrial library.

For these circuits we executed several iterations of *dch* and *strash* commands before applying ARTi to eliminate extra logic introduced for meeting timing constraints. As can be seen from the tables, our algebraic rewriting is significantly more efficient than those using computer algebra, GB-reduction based approach.

Table 3.1: CPU verification time (in seconds) of synthesized and technology mapped multipliers using different libraries. #GT = Number of gate types. FI≥5 = Number of gates with fanin≥5.

Designs	ARTi	#GT	FI≥5	[66]	[65]
btor64-resyn3-nomap	0.1	-	-	711	4.2
abc64-resyn3-nomap	0.1	-	-	801	4.0
btor128-resyn3-nomap	0.3	-	-	ES	ES
abc128-resyn3-nomap	0.1	-	-	ES	ES
btor64-resyn3-map-simple	0.3	7	0	1073	418
abc64-resyn3-map-simple	0.1	7	0	1071	415
abc128-resyn3-map-simple	1.8	7	0	ES	ES
abc64-resyn3-map-14nm	29	15	17	TO	TO
abc64-resyn3-map-7nm	MO	24	9,791	TO	TO
abc128-resyn3-map-14nm	400	15	1,008	ES	ES
abc128-resyn3-map-7nm	MO	23	26,600	ES	ES

Table 3.2: CPU verification time (in seconds) for multipliers prior to synthesis. ES = Error State reported by Singular.

Design	ARTi	[66]	[65]
btor-16	0.01	0.5	0.01
btor-32	0.02	11.7	0.3
btor-64	0.1	725	4.0
btor-128	0.5	ES	ES
sp-ar-rc16	0.01	1.1	0.01
sp-ar-rc32	0.1	35.5	0.3
sp-ar-rc64	0.4	1312	4.6
sp-ar-rc128	1.6	ES	ES
abc-256	0.7	ES	ES
abc-512	3.7	ES	ES

3.6 The Bit-flow Model

This section offers a new insight into an arithmetic circuit verification problem, in which the computation performed by the circuit is treated as the *flow* of digital data. The goal here is not to introduce any new algorithms, but to suggest an interpretation how the computation propagates in an arithmetic circuit. This interpretation will then provide an argument for soundness and completeness of the algebraic rewriting method, independently from the computer algebra arguments.

The circuit is modeled as an *acyclic network* of logic and/or arithmetic components connected via electrical *signals* or wires. Mathematically, the signals are represented as *variables*, denoted X; they include the internal signals, the primary inputs (PI), and the primary outputs (PO). The terms *signals* and *variables* will be used interchangeably, depending on the context (structural vs. functional view of the circuit). Each component of the circuit is described by its *characteristic function*, a pseudo-Boolean polynomial function relating the component's inputs to its outputs. The characteristic functions of Boolean logic gates are provided by Equation 6.1. For example, the characteristic function of an OR gate $z = a \vee b$ is $z = a + b - ab$. Similarly, the characteristic function of a half adder (HA) is $2C + S = a + b$, etc.

The generic term *flow* is intuitively understood as a movement of some physical entity (such as current or fluid) through the network. Here, it is a movement of digital data (voltage potentials taking value 0 or 1) whose capacity is measured in bits, where each bit contributes one unit of flow to its value. The flow starts at the primary inputs and propagates towards the primary outputs, distributed internally according to the characteristic functions of the circuit components. For example, a full adder accepts an in-flow of three bits, a, b, c and "distributes" this flow to the outputs according to its characteristic function: $a + b + c = 2C + S$. The coefficient associated with each variable represents its "capacity", the maximum value of the flow that can pass through the corresponding signal. In a half-adder or a full-adder,

the weight of each input bit is 1, and the weight of the output bits C and S are 2 and 1, respectively. For a logic gate, the inputs and the output bits have a weight of 1 each.

The idea of using the *flow conservation law* to verify arithmetic circuits has already been proposed in [21]. However, it is applicable there only to arithmetic circuits composed of half- and full-adders, where the circuit elements and the specification are modeled as linear expressions. Here, we extend this idea to an arbitrary integer arithmetic circuit which computes an arithmetic function as a polynomial.

The value of the flow in the circuit is captured by the polynomials (signatures) generated during the algebraic rewriting. Equations (3.6) and (3.7) are examples of such polynomials. The value of the flow at the primary inputs is represented by the specification polynomial F_{spec}, while the value of the flow at the primary outputs is represented by the output signature S_{out}. The value of the flow at an arbitrary *cut* of the circuit (defined below) is represented by a polynomial in terms of the variables associated with the respective signals of the circuit. It can be computed from the polynomial generated at each step of the algebraic rewriting. We shall show that the *value* of the flow in an arithmetic circuit represented by such polynomials is *invariant* throughout the circuit.

In principle, the circuit can be composed of arbitrary components, with single-output logic gates as well as multiple-output arithmetic modules, such as half- and full-adders; or any module for which the I/O relationship can be defined as a polynomial. Here we limit our attention to gate-level arithmetic circuits with single-output logic gates. In the remainder of this section, any reference to polynomials S_i, S_{in}, S_{out} or F_{spec} assumes that they are reduced over the field polynomials $< x^2 - x >$, which is implicitly achieved by replacing x^2 with x during the algebraic rewriting (refer to Section 3.3). It should be clear that the value of the flow is not affected by

this transformation or by any simplification which removes the terms that evaluate to zero, since it does not change the value of the polynomial.

Consider a polynomial P_i generated at step i of the algebraic rewriting process. It can be observed that the variables X_i that are in the support set of P_i correspond to a *cut* in the circuit. Using network flow terminology, the *cut* is a set of signals that partitions the circuit into two subsets: one containing the gates whose inputs are transitively connected to the primary inputs PI, and the other containing the gates whose outputs are transitively connected to the primary outputs PO. This separation is an inherent property of backward rewriting: starting with the output signature polynomial $P_i = S_{out}$, a variable $x_k \in X_i$ of P_i that represents an output of some gate g_k is replaced by the polynomial in its inputs. From the structural viewpoint, this moves the cut from the gate output to its inputs. From this perspective, the polynomial P_i can also be viewed as a *signature* of the cut C_i, denoted S_i.

Polynomial expressions in Eqn. (3.6) and (3.7) are examples of cut signatures for the full adder circuit of Figure 3.1. The input and output signatures, S_{in} and S_{out} defined earlier, are the signatures of the boundary cuts, associated with the primary inputs PI and primary outputs PO, respectively. The following example illustrates the relationship between the polynomial and cut rewriting.

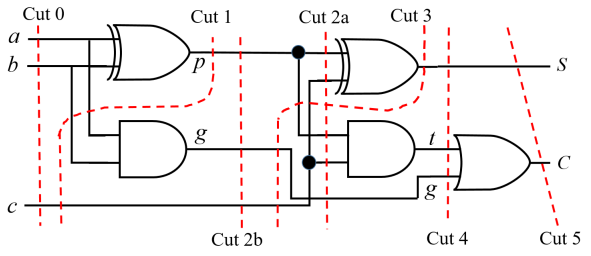


Figure 3.3: Cut rewriting in a full-adder circuit.

Example 1: Figure 3.3 shows a full adder circuit (FA) with a set of cuts. The signatures $\{S_{out}, S_4, S_3, S_2, S_1, S_{in}\}$, associated with cuts $\{Cut_5, ..., Cut_0\}$, are given in Eqn. 3.9. They are obtained by successively rewriting the output signature $S_{out} =$

$2C + S$ of Cut_5 through the circuit. Specifically, the signature S_{out} is transformed into signature S_4 of Cut_4 by replacing variable C with the expression of the OR gate, $C = g + t - gt$, resulting in the signature $S_4 = 2(g + t - gt) + S$. This signature is then transformed into S_3 by rewriting across the AND gate, $t = cp$, etc., until it reaches the primary inputs. The following signatures are obtained by successive rewriting of the circuit, in the order consistent with the ordering rules discussed in Section 3.3. Furthermore, the expression for S_3 is reduced here by applying XOR-AND simplification rule of [68], namely $pg = 0$.

$$\begin{aligned} S_{out} &= 2C + S \\ S_4 &= 2(g + t - gt) + S \\ S_3 &= 2(cp + g - cpg) + S \\ &= 2(cp + g) + S \\ S_2 &= c + p + 2g \\ S_1 &= c + p + 2ab \\ S_{in} &= c + a + b \end{aligned} \tag{3.9}$$

Note that, in contrast to the network flow model of [21], the signature S_i of some cut C_i is not a linear combination of its signals X_i, but in general a nonlinear polynomial S_i in variables X.

We now introduce the notion of the *flow value*, a measure of the capacity of the bit-flow across a cut.

Definition 1: The *value* of a cut C_i with signature S_i for a given assignment of variables X_i is the value of its signature S_i evaluated at X_i. It is denoted as $V(S_i)(X_i)$.

One should keep in mind that the values of variables X_i of a cut cannot be arbitrary but can assume only those values that can be derived from the bit values of PI. To this effect, we introduce the following definition.

Definition 2: The assignment of variables in X_i is called *legal*, denoted by $[X_i]$, if it is derived from an assignment of the primary inputs, X_{PI}. In this case we say that $[X_i]$ is *compatible* with X_{PI}.

With this we will use the notation $V(S_i)[X_i]$ to denote the value of the cut only for *legal* assignment of X_i. We can then say that two assignments, $[X_i], [X_j]$, are *compatible* if they are both derived from the *same* values X_{PI}.

The reason for introducing the concept of legality is that one can only reason about the flow through the cuts for only those values of signals that are actually generated by the circuit.

Example 2: Table 3.3 shows the flow values for the FA circuit in Figure 3.7 at each cut for all possible PI assignments. These values are obtained by simply substituting given values of $[X_i]$ into the expression of S_i.

Table 3.3: Flow values of cuts in the correct circuit. $S_5 = S_{out} = 2C + S$; $S_0 = S_{in} = a + b + c = F_{spec}$

PIs			Intermediate			*POs*		Flow value $V(S_i)$ at Cut_i						
c	a	b	p	g	t	C	S	S_5	S_4	S_3	S_2	S_1	S_0	F_{spec}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
0	0	1	1	0	0	0	1	1	1	1	1	1	1	**1**
0	1	0	1	0	0	0	1	1	1	1	1	1	1	**1**
0	1	1	0	1	0	1	0	2	2	2	2	2	2	**2**
1	0	0	0	0	0	0	1	1	1	1	1	1	1	**1**
1	0	1	1	0	1	1	0	2	2	2	2	2	2	**2**
1	1	0	1	0	1	1	0	2	2	2	2	2	2	**2**
1	1	1	0	1	0	1	1	3	3	3	3	3	3	**3**

An important observation is that, for a given assignment of X_{PI}, the values of all cuts (and of their signatures) are invariant.

Definition 3: Two cuts, C_i, C_j, with signatures S_i, S_j, are *congruent*, denoted $C_i \cong C_j$, if for every pair of compatible assignments, $[X_i], [X_j]$, their values are the same, i.e., $V(S_i)[X_i] = V(S_j)[X_j]$. In this case, we also say that the corresponding signatures are congruent, denoted $S_i \cong S_j$.

Theorem 1: Given a pair of cuts C_i, C_j, such that C_i is transformed into C_j or, equivalently, S_i rewritten into S_j by algebraic rewriting, the two cuts are congruent. That is $S_i \longrightarrow S_j \implies S_i \cong S_j$.

Proof. A cut $C_i(X_i)$ can be transformed into another cut $C_j(X_j)$ by a series of algebraic rewriting transformations over logic gates, each described by some polynomial $g = v - tail(g)$. During rewriting, every occurrence of variable v in the source cut (initially C_i) is replaced by $tail(g)$ in the target cut (finally C_j). Since polynomial g satisfies the relation $g = v - tail(g) = 0$, provided by Eq. (1), then $v = tail(g)$. Consequently, $V(S_i) = V(S_j)$ for all values of variables v and those in $tail(g)$ that satisfy this relation. Hence, $V(S_i)[X_i] = V(S_j)[X_j]$ for all compatible assignments $[X_i], [X_j]$, and thus by Definition 6 they are congruent, $S_i \cong S_j$. □

Example 3: Theorem 1 states an important property of bit-flow conservation across the cuts in an arithmetic circuit. Table 3.3 gives the values of individual cuts for the full-adder circuit in Figure 3.3. As we can see, the signature value of each cut in the original (correct) circuit, including the inputs and output signatures are the same for all primary input assignments.

Notice that two cuts may be congruent even if one cannot be obtained from the other by rewriting. For example, in Figure 3.3, $Cut_3 = \{S, c, p, g\}$ and cut $\{p, c, t, g\}$ (crossing each other, not shown in the figure) cannot be derived from each other since there are no gates that can transform one into another; yet, they are also congruent since each can be derived by a rewriting of S_{out}, albeit through a different set of gates. To that effect, we have the following Corollary:

Corollary 1: All cuts in the circuit are mutually congruent. In particular, $S_{out} \cong S_{in}$.

Proof. By Theorem 1, any cut C_i in the circuit is congruent with the cut at the primary outputs, PO, because it can be obtained by backward rewriting from PO. Any other cut, C_j, is also congruent to PO. That is, by the definition of congruence,

$V(S_i)[X_i] = V(S_{PO})[X_{PO}]$ and $V(S_j)[X_j] = V(S_{PO})[X_{PO}]$, and hence $S_i \cong S_j$, for any cuts C_i, C_j, including S_{in} and S_{out}. As a result, all the cuts are congruent and form an equivalence class. □

Corollary 1 basically states that the value of the flow measured at *any* cut in the circuit is constant throughout the circuit.

We now need to discuss how to distinguish a circuit that is functionally correct from the circuit that is faulty. The circuit is said to be functionally correct if its implementation satisfies the specification; or, equivalently, that the values computed by the circuit are the same as those provided by the specification for all possible input assignments. Using the terminology of algebraic rewriting we can formalize this definition as follows:

Definition 4: The circuit is *functionally correct* if, for each primary input assignment, X_{PI}, the result encoded in the primary outputs X_{PO} satisfies the condition $V(S_{out})[X_{PO}] = V(F_{spec})[X_{PI}]$.

The following theorem specifies the sufficient and necessary condition for the functional correctness of a circuit.

Theorem 2: The circuit is functionally correct if and only if the *input signature*, S_{in}, computed by algebraic rewriting of the output signature, S_{out}, is the same as the *functional specification*, i.e., if $S_{in} = F_{spec}$.

Proof. The *if* part (soundness): let $S_{in} = F_{spec}$, which implies that $V(S_{in}) = V(F_{spec})$ for all possible primary input assignments, X_{PI}. Since, by Corollary 1, $S_{in} \cong S_{out}$, i.e., $V(S_{in}) = V(S_{out})$, we have $V(S_{out}) = V(F_{spec})$ for all possible values of X_{PI}. That is, the circuit is functionally correct.

The *only if* part (completeness): Let the circuit be functionally correct, i.e., $V(S_{out}) = V(F_{spec})$ for all values of X_{PI}. Since $S_{out} \cong S_{in}$, we have $V(S_{in}) = V(F_{spec})$ for all the assignment of inputs X_{PI}. This in turn implies that $S_{in} = F_{spec}$. Furthermore, the rewriting procedure always terminates: the circuit as a DAG has no

loops and the number of rewriting steps is equal to the number of gates. Hence, the method is also complete.

It should be emphasized that the above argument is only valid for pseudo-Boolean polynomials, reduced over field polynomials J_0. It is known that such polynomials have unique polynomial representation, so that two polynomials will evaluate to the same value only if they are the same. □

Example 4: To illustrate the case of a faulty circuit, where $S_{in} \neq F_{spec}$, consider again the full adder example in Figure 3.3 in which the AND gate $g = ab$ has been replaced with an OR gate, $g = a + b - ab$. This causes the signatures of the cuts to change, as follows (note that in this circuit the AND-XOR simplification $pg = 0$ does not apply):

$$\begin{aligned} S_{out} &= 2C + S \\ S_4 &= 2(g + t - gt) + S \\ S_3 &= 2(cp + g - cpg) + S \\ S_2 &= c + p + 2g - 2cpg \\ S_1 &= c + p + 2(a + b - ab) - 2cp(a + b - ab) \\ S_{in} &= c + 3(a + b) - 4ab - 2c(a + b - 2ab) \end{aligned} \tag{3.10}$$

The input signature obtained by this rewriting is now: $S_{in} = c + 3(a + b) - 4ab - 2c(a + b - 2ab)$, which does not match the circuit specification, $F_{spec} = a + b + c$. The flow values for each cut, for each assignment X_{PI}, are shown in Table 3.4. The table confirms that all the cuts $\{S_5, S_4, S_3, S_2, S_1, S_0\}$ are congruent; and the flow value at any of the cuts, according to Theorem 1, is constant for any PI assignment. However, the flow value for some assignments of X_{PI} is different than in the correct circuit (shown in column F_{spec}), proving that the circuit is faulty.

In summary, in the circuit that computes a polynomial, the value of the flow from PI to PO is *constant* throughout the entire circuit. In the functionally correct circuit the value of the flow equals that of F_{spec}; in a faulty circuit the flow value is different

Table 3.4: Flow values in faulty circuit (gate AND of g replaced by OR); $S_5 = S_{out} = 2C + S$; $S_0 = Sin \neq F_{spec}$

PIs			Intermediate			POs		Flow value $V(S_i)$ at Cut_i						
c	a	b	p	g	t	C	S	S_5	S_4	S_3	S_2	S_1	S_0	F_{spec}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
0	0	1	1	1	0	1	1	3	3	3	3	3	3	**1**
0	1	0	1	1	0	1	1	3	3	3	3	3	3	**1**
0	1	1	0	1	0	1	0	2	2	2	2	2	2	**2**
1	0	0	0	0	0	0	1	1	1	1	1	1	1	**1**
1	0	1	1	1	1	1	0	2	2	2	2	2	2	**2**
1	1	0	1	1	1	1	0	2	2	2	2	2	2	**2**
1	1	1	0	1	0	1	1	3	3	3	3	3	3	**3**

than that of F_{spec}, while all the cuts remain congruent. If the circuit is correct, S_{in} will match the specification, F_{spec}; otherwise, the algorithm will report the circuit as faulty and will return the computed signature S_{in}.

CHAPTER 4

VERIFICATION OF TRUNCATED ARITHMETIC CIRCUITS

In this chapter, we present an approach for formal verification of truncated multipliers that perform approximate integer multiplication. This method is based on extracting polynomial signature of the truncated multiplier using algebraic rewriting, discussed in Chapter 3. To efficiently compute the polynomial signature, a multiplier reconstruction is proposed to construct the complete, precise multiplier from the truncated one. This method has been tested on multipliers up to 256 bits with three truncation schemes: *Deletion*, *D-truncation*, and *Truncation with Rounding*. Experimental results are compared with the state-of-the-art SAT, SMT, and computer algebraic solvers.

4.1 Problem Statement

The traditional verification methods discussed in Chapter 1.2 are not applicable to large arithmetic circuits as they require "bit-blasting", causing memory overload. Even more so, they also fail when dealing with large truncated arithmetic circuits. Even the state-of-the-art SAT solvers cannot solve verification problem of a 16-bit truncated multiplier.

The computer algebra approaches discussed in Chapter 3 can successfully verify large arithmetic circuits, such as 512-bit multipliers. However, when verifying truncated arithmetic circuits, in particular truncated multipliers, these methods face serious memory problem. Even for verifying a 8-bit truncated multiplier, the system

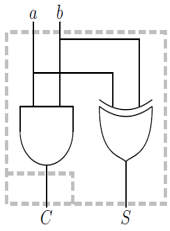


Figure 4.1: Complete half adder.

runs out of 16 GB memory with an inconclusive result. This is because truncation prevents polynomial cancellations and simplifications that naturally occur during the rewriting process [86]. For example, a complete half-adder (HA) in Figure 4.1 consists of an XOR for the sum and an AND for the carry, while the truncation process makes some of the HAs in the original circuit incomplete (with the XOR or the AND gate removed). The algebraic models of XOR ($a \oplus b = a + b - 2ab$) and AND ($a \cdot b = ab$) gates of the HA are nonlinear, but when they are added together with correct weights, the resulting polynomial becomes linear: $2C+S = 2\cdot(ab)+(a+b-2ab) = a+b$. This naturally occurs during HA rewriting in a "complete" arithmetic circuits. Whereas in the case of a truncated circuit, some of the output bits are truncated, and the corresponding gates (mainly XORs) computing those bits will be removed. The above linear relationship is then broken, and the non-linear term ab remains and gets expanded by further rewriting of variables a and b in the polynomial expression. This causes an exponential increase in the size of intermediate polynomials, resulting in memory explosion.

To address this issue, a multiplier reconstruction approach is used to construct a complete, precise multiplier from the truncated one. The method consists of three basic steps: 1) determine the weights (binary encoding) of the output bits; 2) reconstruct the truncated multiplier using functional merging and re-synthesis; and 3) construct the polynomial signature of the resulting circuit. The details are discussed in Section 4.3.

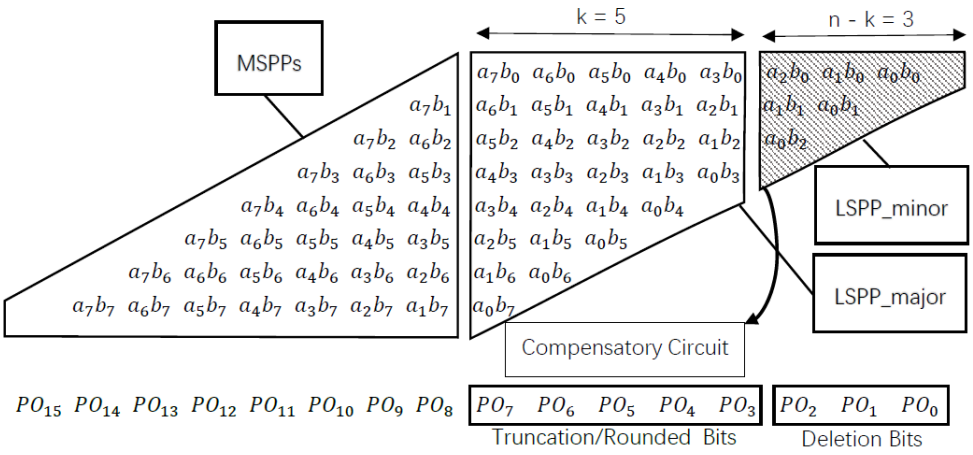


Figure 4.2: Partial product array of a 8-bit multiplier.

4.2 Formal Truncation Schemes

Three formal truncation schemes used to design truncated multipliers are: *Deletion*, *D-truncation* and *Rounding*. An example of truncated multiplier is shown in Fig. 4.2. Normally, a $2n$-bit output is generated for a regular $n \times n$ multiplier, whereas in this case only n most significant bits (MSBs) are computed. A multiplier that has the same bit-width in their input and output is called a *fixed-width multiplier*, which is mainly used in this chapter. In this example, the inputs $a[n\text{-}1{:}0]$ and $b[n\text{-}1{:}0]$ are assumed to be two integer inputs. The partial products are divided into two subsets:

1. The most significant partial products (*MSPPs*), corresponding to n MSBs; and,

2. The least significant partial products (*LSPPs*), further subdivided into *LSPP_major* and *LSPP_minor*, corresponding to the k most significant columns of LSPPs and the remaining (nk) columns, respectively.

In Deletion scheme, the partial products in *LSPP_minor*, which have a small impact on the accuracy of the result, are discarded. The corresponding PPs are dropped to prevent the carry chain from propagating further. This decreases the size of the carry chain, which reduces the delay and power (the main reason for applying truncation). Even though *LSPP_minor* have a smaller contribution to the accuracy of the MSBs,

the error of discarding them can be high; in the worst case it is $7 \cdot 2^8$ for an 8-bit fixed-width multiplier. To rectify this, a compensatory circuit is typically added after deletion [60][75].

In D−truncation scheme, some output bits computed by LSPPs are truncated without modifying the entire carry chain. In contrast to the Deletion scheme, which improves performance and saves power, D-truncation does not change the functionality of the remaining circuit. Instead, it only removes these truncated bits and the associated gates. The purpose of D-truncation is to manage the number of output bits, i.e., maintaining the number of input and output bits to be the same. Although D-truncation will not affect the accuracy of the MSBs, the error due to losing the information in LSPPs is $(2^8 - 1)$ in the worst case.

Rounding scheme is often introduced to achieve further accuracy and truncation [31]. Instead of truncating all columns in *LSPP_major*, some of the most significant columns in *LSPP_major* are kept for the rounding circuit. The value of those columns in *LSPP_major* is rounded into MSPPs.

4.3 Verifying Different Truncation Schemes

In this section, we describe two basic techniques, *functional merging* and *re-synthesis*, used to verify truncated multipliers. We introduce a Partial Product Detector (PPD) which assigns correct weights to the respective output bits. In order to better understand how PPD works, we analyze in Section 4.3.1 a truncated multiplier designed with only the Deletion scheme. In Section 4.3.2, we consider a truncated multiplier designed with only the D-truncation scheme and describe the details of functional merging and re-synthesis. Lastly, we propose a comprehensive verification analysis where all truncation schemes are combined together. Fig. 4.4 shows the verification flowchart of our methodology applicable to a general truncated multiplier. Two assumptions are made:

- The compensatory circuits for Deletion and Rounding can be extracted from the high-level synthesis netlist, and represented as polynomials in the primary inputs.

- The bit position of primary inputs ($a[n\text{-}1:0]$, $b[n\text{-}1:0]$) is known, so that the weights of the partial products are also known.

4.3.1 Deletion Scheme only

In the Deletion scheme, some of the partial products are removed to reduce the circuit complexity at the expense of accuracy. For example, in Fig. 4.2, six of the partial products in *LSPP_minor* are removed, and the logic columns PO_0, PO_1 and PO_2 in the shaded area will disappear. As a result, there is no carry feed into the PO_3 column. Hence, the remaining adder tree still executes complete additions (the remaining HA/FA blocks are complete). In this case, there is no polynomial size explosion, since all additions preserve polynomial eliminations during the rewriting process. This means that function extraction is sufficient to extract the input signature of the circuit.

In Fig. 4.2, when the logic cones associated with *LSPP_minor* columns are removed, PO_3 corresponds to the current LSB. Hence, the weight of the current LSB that was 2^3 in the exact multiplier is shifted to 2^0. The computed input signature will never match the specification until all the output bits are assigned their original weight. Moreover, additional polynomials representing those removed partial products must be added. In this case, PPD performs the following functions (Algorithm 3):

- Assign the correct/original weight to each current output bit.

- Detect the deleted elements in *LSPP_minor* and generates additional polynomials.

Algorithm 3 Weight Determination and Signature Generation

Input: Gate-level netlist of truncated multiplier
Output: Correct weight of each output bit
Output: Additional polynomials representing the removed partial products

$\mathcal{PO}$=$\{PO_0, PO_1, ..., PO_c\}$: current output bits of truncated multiplier
for $i \leftarrow 0$ **to** $2n-1$ **do**
    Create list$_i$ corresponding to logic column i
    for $j \leftarrow 0$ **to** i; $k \leftarrow 0$ **to** i; $j+k \leftarrow 0$ **to** i **do**
        Search partial product $a_j b_k$ in the netlist
        if partial product $a_j b_k$ exists **then**
            Save product $a_j b_k$ into list$_i$
        else Additional Polynomials $\leftarrow$ Additional Polynomials + $2^i a_j b_k$
        end if
    end for
    Number_of_Total_PPs $\leftarrow$ *Number_of_Total_PPs* + length(list$_i$)
    Rank_of_Logic_Column $\leftarrow i$
    Save Pair$_i$(*Rank_of_Logic_Column*, *Number_of_Total_PPs*)
end for
for $i \leftarrow 0$ **to** c **do**
    Search and count the number of PPs that current output bit PO_i depends on
    Match the count with *Number_of_Total_PPs* in Pairs
    return *Rank_of_Logic_Column*
    Assign correct weight $W_i \leftarrow 2^{Rank\_of\_Logic\_Column}$
end for
return Correct weights $\mathcal{W}$=$\{W_0, W_1, ..., W_c\}$ and Additional Polynomials

In general, for an $N_a \times N_b$-bit multiplier, Algorithm 3 will search $N_a + N_b$ logic columns. Specifically, for a 8-bit multiplier shown in Fig 4.2, it searches $2 \times 8 = 16$ logic columns, PO_0 to PO_{15}. The results of applying Algorithm 3 to this example are shown in Table 4.1.

Table 4.1: The relationship between *RLC* and *NTPP*.
RLC: Rank of Logic Column, *NTPP* : Number of Total PPs

RLC	*NTPP*	*RLC*	*NTPP*	*RLC*	*NTPP*
0	0	5	15	10	48
1	0	6	22	11	52
2	0	7	30	12	55
3	4	8	37	13	57
4	9	9	43	14,15	58

To assign the correct weight for each output bit, we traverse and count all those PPs on which that output bit depends. This information is stored in Table 4.1, which shows for each output bits (rank of logic column, *RLC*) the number of total PPs (*NTPP*) that it depends on. Since all PPs in the *LSPP_minor* logic were removed, *NTPP* for $RLC = 0, 1, 2$ is 0. To find the weight of the current LSB of this truncated

multiplier, we examine the PPs that this bit depends on. In this case, four PPs have been found, namely $a_0b_4, a_1b_3, a_2b_2, a_3b_1, a_4b_0$. We then examine Table 4.1 to find which bit (RLC) depends on $NTPP = 4$. It turns out that $RLC = 3$ in this case, which means the current LSB of this truncated multiplier is the PO_3 of the complete multiplier. Hence, the correct weight for this bit must be 2^3.

In the same way, we can assign the correct weights to the other output bits. For example, if we determine that one of the output bits depends on 15 PPs, then the correct weight of that output bit is 2^5, since RLC for $NTPP = 15$ is 5. Furthermore, those 15 partial products consist of 6 PPs in PO_5 logic column, 5 PPs in PO_4 and 4 PPs in PO_3.

Algorithm 3 explains a general idea of how the PPD works. However, a special case needs to be discussed. Notice that the two MSBs PO_{14} and PO_{15} depend on the same number of partial products, that is 58, as shown in Table 4.1. In such a one-to-two mapping problem, PPD will determine the correct weight according to the information of the circuit. That is, the output bit with the higher rank is assigned the higher weight, that is 2^{15}, and the other output bit as 2^{14}.

Once all the output bits have been assigned their correct weights, we can then apply function extraction to compute the input signature. Notice that the boundary between *LSPP_minor* and *LSPP_major* does not necessarily have to be straight (along the column lines). As long as the correct weight of each output bit is assigned and the *LSPP_minor* has been detected, the calculated signature will always be correct.

While searching each logic column, the discarded/undetected partial products will be added as additional polynomial at the end of the signature calculation, with the corresponding coefficients. In this case, the additional polynomial will be $2^2(a_2b_0 + a_1b_1 + a_0b_2) + 2^1(a_1b_0 + a_0b_1) + 2^0a_0b_0$. The polynomial of the compensatory circuit D also need to be considered, if it exists. The size of the compensatory circuit depends upon the number of partial products removed. The worst case size occurs when $k = 0$,

that is, when all the LSPPs are removed. However, the bigger the compensatory circuit, the harder the verification process is. In the worst case scenario, most of the time is spent on computing D. Therefore, if the size of the compensatory circuit is too big, we need to remove it before the rewriting process.

Finally, we compare: 1) the calculated polynomial of the truncated circuit expanded by additional polynomial, with 2) the sum of full multiplier specification and polynomial D. The reason for doing this, is that the compensatory circuit is included in the truncated circuit. This operation is trivial when there is no compensatory circuit ($D = 0$). If the two polynomials are equal, the given circuit is proved to be a Deletion-based truncated multiplier.

4.3.2 D-truncation scheme only

D-truncation is the process in which some of the output bits are removed without changing the functionality of the remaining output bits. A fixed-width multiplier can be built by simply applying the D-truncation scheme to a regular multiplier. For example, in Fig. 4.2, assuming that $k = 8$, no partial product is removed and 8 LSBs are truncated. Although the functionality of the remaining 8 MSBs has not changed, the entire structure of adder tree is no longer complete. For example, assume that one output bit is computed by the XOR gate of a HA block. When this bit is truncated, the XOR gate will be automatically removed from the circuit because it is redundant. However, the AND gate will remain to preserve the carry chain.

In this example, the XOR gates directly connected to PO_2 to PO_7 will disappear due to D-truncation. The remaining AND gates make the structure of adder tree incomplete. The nonlinear terms of polynomials, which were supposed to be canceled, will propagate during the backward rewriting. This leads to an exponential increase in memory size, potentially ending up in a memory explosion. In other words, we cannot directly apply function extraction to such a circuit. To efficiently apply function

extraction, our approach transforms the incomplete function into a complete one. This is achieved by functional merging, followed by re-synthesis of the combined circuit by ABC [50].

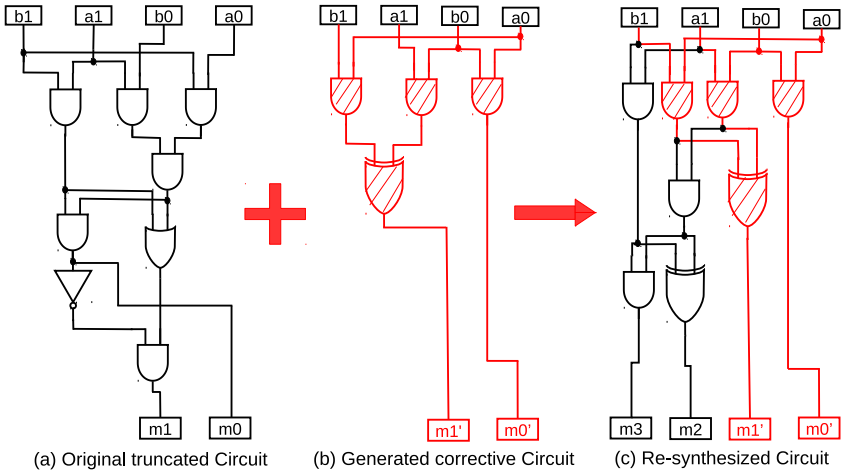


Figure 4.3: Functional Merging and Re-synthesis.

To better explain this idea, we use a 2×2 multiplier in Fig. 4.3 as an illustrative example. A 2-bit truncated multiplier (with two LSBs truncated) is presented in Fig. 4.3(a). Assume that Fig. 4.3(a) represents the circuit to be verified if it is a 2-bit fixed-width multiplier. We use Algorithm 1 to determine the correct weights of this circuit. After reading the gate-level netlist, PPD assigns weight 2^3 to $m1$ and 2^2 to $m0$. Then, we generate a regular 2×2 multiplier and remove the output bits with weights 2^2 and 2^3. The generated circuit is shown in Fig. 4.3(b), whose weight of $m1'$ is 2^1 and of $m0'$ is 2^0. Finally, we apply our scheme of functional merging and re-synthesis resulting in the circuit in Fig. 4.3(c). The red (shaded) part in this figure corresponds to the generated corrective circuit of Fig. 4.3(b).

If the re-synthesized circuit proves to be a 2×2 multiplier by algebraic polynomial rewriting, this means that the given circuit in Fig. 4.3(a) is indeed a truncated multiplier. This is because the circuit generated in Fig. 4.3(b) has a correct function of two LSBs in a regular 2×2 multiplier. If, and only if, the circuit in Fig. 4.3(a) has

the function of two MSBs of a regular 2×2 multiplier, the re-synthesized circuit can be a 2×2 multiplier.

In general, for a given circuit X that needs to be verified whether it is the D-truncation-based multiplier or not, we first calculate the correct weights for each output bit by PPD. Then, we generate a regular multiplier and truncate those output bits that have the same weight as the ones in circuit X. The generated circuit should have the same number of input bits as circuit X. After functional merging and re-synthesis, the number of output bits in the re-synthesized circuit is equal to the sum of output bits in the given circuit X and the generated circuit, as shown in Fig. 4.3. Finally we apply function extraction (backward rewriting) to the re-synthesized circuit. If the computed signature matches the specification of a regular multiplier, the functionality of circuit X as truncated multiplier is confirmed.

The goal of functional merging is to find the maximum functional similarity between the given circuit and the generated corrective circuit. The merging eliminates as much as possible incomplete addition in the given circuit to speed up function extraction. Resynthesis (performed using ABC [50]) transforms the two circuits into *And-Inverter-Graphs (AIGs)* and performs sweeping, redundancy removal, common logic identification, etc. If the two circuits (the original one and the reconstructed one) have been built in the same way, resynthesis in effect transforms the structure of the original circuit into the circuit performing a complete arithmetic function.

Despite ABC being the state-of-the-art synthesis and verification tool, it could not solve the equivalence checking problem for two different 12-bit truncated multipliers. In contrast, our approach is scalable and can be used to verify truncated multipliers up to 256 bits. The success of such verification depends on the local structural similarity between the original and the generated corrective circuit. Such a similarity assumption is acceptable, especially in industry. For example, designer of the fixed-width multiplier may have access to the original regular multiplier. If the

designer use the structure of the regular multiplier to build the generated circuit, as in Fig. 4.3(b), the problem can be solved very fast. This is also true even if the original circuit and the generated circuit are not built using the same algorithm but exhibit enough local functional similarity. The speed of applying function extraction to the re-synthesized circuit will be much higher than directly applying function extraction to the original circuit. This is because more nonlinear terms will be canceled during polynomial rewriting. However, one cannot reconstruct a circuit without considering the algorithm that was used to design it. For example, functional merging between a Booth-Multiplier and non-Booth multiplier may make the process fail.

Another significant contribution of this approach is that we can verify a multiplier whose arbitrarily output bits have been truncated. For example in Fig. 4.3(a), PPD can assign the correct weight to each of the output bits. We can then generate a regular 2×2 multiplier and truncate those output bits that have the same weight as the one in the given circuit. Therefore, we don't need to know which output bits are truncated. Instead, we can automatically detect arbitrarily truncated bits and follow the same procedure as described earlier.

A special case in weight determination is when either the MSB or (MSB-1)th bit has been truncated. Since these two bits depend on all the PPs, PPD cannot tell which one is truncated. In this case, we will generate two circuits: one circuit assuming that MSB is truncated, and the other circuit assuming that (MSB-1)th bit is truncated. In this case, both circuits are reconstructed, and only if either of them turns out to be a regular multiplier, we conclude that the given circuit is a truncated multiplier.

4.3.3 Deletion + D-truncation + Rounding

In this section, we discuss the case when a truncated multiplier is designed using all of the three truncation schemes, Deletion, D-truncation, and Rounding. Partial

product detector (PPD), functional merging, and re-synthesis can still be used along with the function extraction to solve the verification problem. Fig. 4.4 shows the verification flow of our methodology applicable to the Deletion only, D-truncation only, and a combination of these two schemes.

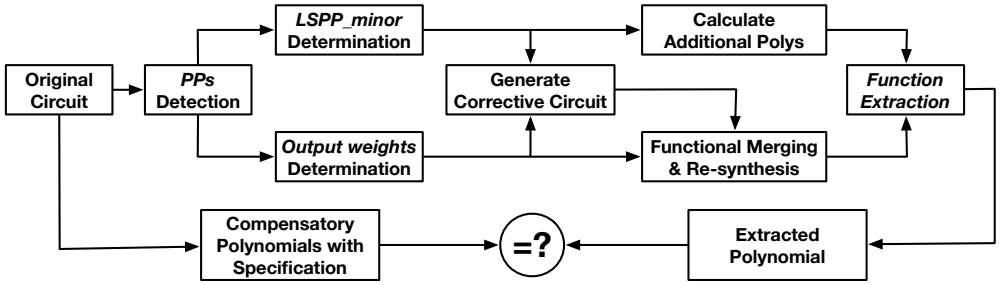


Figure 4.4: Verification Flow dealing with all truncation schemes.

An 8-bit fixed-width multiplier is used as an example for the analysis of these truncation schemes. Assume that six of the partial products in *LSPP_minor* are removed, as shown in Fig. 4.2. Therefore, the output bits associated with PO_0, PO_1, and PO_2 will disappear due to Deletion. Then, five output bits, corresponding to logic columns PO_3 to PO_7, will be removed using D-truncation and Rounding scheme.

In the verification flow, PPD detects the elements in *LSPP_minor* and assigns a correct weight to each of the output bits. Then, we generate a regular multiplier which has the same number of inputs as the original circuit. An additional step, which is different from the analysis in Section 4.3, is as follows. We discard the undetected PPs in the generated circuit so that the PPs in the generated circuit and the PPs in the original circuit are the same. After that, the generated circuit has 5 output bits, corresponding to PO_3 to PO_7, while the original circuit has 8 output bits, corresponding to PO_8 to PO_{15}. We then apply functional merging and re-synthesis to the combined circuit.

Finally, we calculate the polynomials as shown in Fig. 4.4, and compare the following two terms: 1) the computed signature of the re-synthesized circuit with additional polynomials representing the undetected PPs (*LSPP_minor*); with 2) the specifica-

tion of the regular 8-bit multiplier enlarged by the polynomials of the compensatory circuits. Upon being equal, the given circuit is proved to be an 8-bit fixed-width multiplier. Experiments were performed on a Baugh-Wooley multiplier and a CSA array multiplier for up to 256 bits.

4.4 Results

The partial product detector (PPD) in the proposed method has been implemented in Python. ABC, a popular synthesis and verification tool [50] was used to implement the functional merging and re-synthesis. We performed our experiments on an Intel Core CPU i5-3470 @ 3.20 GHz 4 with 15.6 GB memory. Experiments are performed on a Baugh-Wooley multiplier and a CSA multiplier for up to 256 bits. The Baugh-Wooley multiplier is a modified non-Booth unsigned multiplier using Baugh-Wooley scheme. CSA multiplier implemented in our experiments is an array based CSA multiplier, generated by ABC. Unless stated otherwise, the truncated multipliers in our experiments are obtained by removing 1/4 partial products and truncating up to half of the output bits (LSBs).

Table 4.2: Results and comparison with Function Extraction [22] using truncated CSA multipliers.

*TO : Time out of 9,000sec, MO : Memory out of 10GB.

# bits	Function Extraction [22]		This Work	
	Runtime (sec)	Memory (MB)	Runtime (sec)	Memory (MB)
6	38.94	296.0	0.01	4.2
8	1174.4*	*MO*	0.01	4.8
16	928.2*	*MO*	0.05	7.3
32	796.3*	*MO*	0.25	17.2
64	631.4*	*MO*	1.05	57.3
128	470.9*	*MO*	4.40	220.4
256	286.1*	*MO*	18.94	880.1

As analyzed in Section 4.3, a direct application of function extraction to a truncated multiplier causes a memory explosion during polynomial rewriting. Table 4.2 makes a comparison between our scheme and the original function extraction [22] in terms of CPU execution time and memory consumption. Truncated multipliers used

here are CSA-based multipliers. Function extraction [22] succeed only for truncated multipliers with the operand size up to 6. For larger operand sizes the system runs out of memory and results in an incomplete calculation. The runtime numbers labeled with * indicate the CPU time up to the moment when memory usage exceeds 10GB. In principle, given enough memory, function extraction might be able to output a correct result. However, such an experiment is not feasible or scalable. Hence, a direct implementation of function extraction to the truncated multiplier is neither efficient nor scalable. In contrast, our approach based on functional merging and re-synthesis is very fast, since we eliminate the incomplete additions in a truncated multiplier.

Table 4.3: Results and comparison with *ABC, SMT, and SAT* solvers using truncated Baugh-Wooley multipliers.

*TO : Time out of 9,000sec, MO : Memory out of 10GB.

# bits	cec-ABC [50] (sec)	Lingeling [9] (sec)	Boolector [55] (sec)	This Work	
				Runtime (sec)	Memory (MB)
6	0.41	0.30	0.16	0.01	4.6
8	16.2	5.92	3.49	0.28	15.4
10	318.1	350.3	261.6	0.50	19.8
12	*TO*	*TO*	8746.4	0.81	24.1
16	*TO*	*TO*	*TO*	1.12	36.8
32	*TO*	*TO*	*TO*	3.87	50.9
64	*TO*	*TO*	*TO*	14.7	268.2
128	*TO*	*TO*	*TO*	68.2	757.6
256	*TO*	*TO*	*TO*	279.3*	*MO*

We also analyzed truncated multipliers designed with Baugh-Wooley scheme with bit-width varying from 6 to 256 bits. Comparison in Table 4.3 shows that our approach gives a much better performance than ABC and Lingeling SAT solver [9]. ABC runtime exceeds 9,000 seconds in checking equivalence between a 12-bit truncated Baugh-Wooley multiplier and a CSA truncated multiplier generated by ABC. The SAT solver of ABC [50], Lingeling solver [9], and SMT solver Boolector [55] all fail for sizes greater than 12 bits for this experiment.

CHAPTER 5

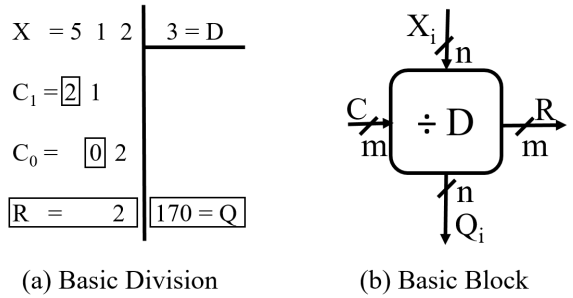
VERIFICATION OF ARITHMETIC CIRCUITS SUBJECTED TO ARITHMETIC CONSTRAINTS

Previous chapter described the reconstruction approach to the verification of arithmetic circuits, in which some of the output bits have been truncated. In this chapter, we will discuss the case when an arithmetic circuit is subjected to arithmetic constraints, applied to its inputs. The concept of vanishing monomial is introduced to enable algebraic rewriting to verify such constrained circuits. A case-splitting verification approach is proposed. The method has been tested on constrained arithmetic circuits, such as adders, multipliers and dividers up to 128 bits.

5.1 Problem Statement

Certain applications require arithmetic circuits to be customized by imposing some constraints to be satisfied by the circuit. When designing such circuits, the circuit performance or its area can be improved by taking the information of these constraints into account. For example, such constraints may ensure that operands of a divider are normalized such that the MSB always has value 1. Similarly, while a general $n \times n$-bit multiplier has $2n$ output bits, the output range is not $[0, 2^{2n} - 1]$ but $[0, 2^{2n} - 2^{n+1} + 1]$. The designer can take this range into consideration when designing circuits driven by a multiplier. Sometimes the constraints are not directly imposed on the circuit, but given as preconditions that it should satisfy.

Another practical example can be found in the basic block of a constant divider [4], shown in Figure 5.1. In each division iteration, the computed remainder R of one



(a) Basic Division (b) Basic Block

Figure 5.1: Division operation and the basic divider block.

block is fed into the next block as input C. In such a design, the relationship $C < D$ should be satisfied, ensuring that the input remainder value C is strictly less than the divisor value D (a known constant). Verification of such circuit has only recently caught some attention of the verification community [43].

Computer algebra approaches discussed in Chapter 3, which can successfully verify large standard arithmetic circuits, seem to be incapable to deal with the verification of such constrained circuits. The main reason is that the computed input signature of a circuit subjected to an arithmetic constraint will become too large to compute or too complex to analyze. In the next section, an example is given to illustrate the difference between two adder circuits, one being constraint-free and the other subjected to some arithmetic constraints.

5.2 Constraint-free Circuit vs. Constrained Circuit

Recall that a pseudo-Boolean function f is an n-ary function $f\colon \mathbb{B}^n \to \mathbb{Z}$, where $\mathbb{B}$ is a set of Boolean variables, and $\mathbb{Z}$ is a set of integers. A pseudo-Boolean constraint (PB-constraint) can be expressed by an equality or inequality relation between a pseudo-Boolean function and an integer. In general, such a constraint has the form

$$\sum_i a_i \cdot \prod_j l_{ij} \ \# \ c \tag{5.1}$$

where the coefficients a_i and the constant c are integers, # is one of the relations $<$, $\leq$, $=$, $\neq$, $\geq$. A literal l_{ij} can be either a Boolean variable x_{ij} or its negation $\overline{x}_{ij}$. Other, more complex constraints, such as max, min, or cardinality constraints, are not under considered in this work. Since the relations $=$ and $\neq$ can be considered as a special case of $\geq$ or $\leq$, we will focus on the inequality relations $\geq$ and $\leq$. In some cases, Equation 5.1 can be simplified to an arithmetic constraint $X \geq c$ or $X \leq c$, where X represents a word-level input vector of the circuit, with the operand size n, so that the constant integer c is in the range of $[0, 2^n - 1]$.

The following example, Figure 5.2, shows a standard 3-bit adder $Z = A + B$, with the operand A having constraint $A \geq 3$ imposed on it. We shall refer to this circuit as a *conditional adder*. For ease of comparison between the original, constraint-free and the constrained circuit, the original circuit in Figure 5.2, with outputs are $\{z'_0, z'_1, z'_2, z'_3\}$ is shown in black. For the sake of illustration, the entire circuit with outputs $\{z_0, z_1, z_2, z_3\}$ has not been optimized, so that a complete structure of the original adder can be seen. In general, it is difficult to separate the constraint-free part after the entire circuit has been synthesized, since the internal structure may change.

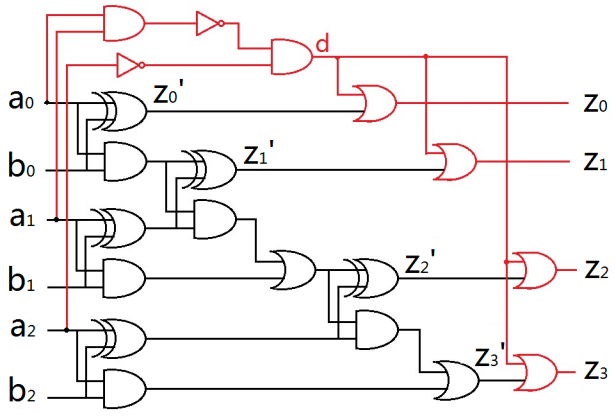


Figure 5.2: conditional 3-bit adder $Z = A + B$, with $A \geq 3$.

Let us start from the unoptimized circuit in Figure 5.2, in which the black part performs a 3-bit addition, stored in the output bits $\{z'_0, z'_1, z'_2, z'_3\}$. To get the function of the entire circuit, we need to explore the relation between $F(z'_0, z'_1, z'_2, z'_3, d)$ and its transformed version, $F(z_0, z_1, z_2, z_3)$, where d is an internal signal shown in the figure. Each output z_k of the transformed circuit is related to the original output by Boolean relation $z_k = z'_k \vee d$ for $k = 0, 1, 2, 3$, where $\vee$ is the Boolean OR operation.

The following is true in the circuit: when $d = 0$, then $z_k = z'_k$, and when $d = 1$, $z_k = 1$, regardless of the value of z'_k. That is, the function of d decides the condition of the circuit. By analyzing the schematic in Figure 5.2, we get $d = \overline{a_2} \cdot \overline{a_0 \cdot a_1} = \overline{a_2 + a_0 \cdot a_1}$, which is exactly in accord with the required condition, $A \geq 3$, imposed on the circuit. If the condition is not satisfied (that is $A < 3$), all the output bits will produce 1, acting like *don't cares*. The function table of the conditional 3-bit adder $A + B$ ($A \geq 3$), shown in table 5.1, are consistent with the expected behavior of the circuit.

Table 5.1: Function table of a conditional 3-bit adder $A + B$ ($A \geq 3$).

$A(a_2a_1a_0)$	$B(b_2b_1b_0)$	d	$Z(z_3z_2z_1z_0)$
000	-	1	1111
001	-	1	1111
010	-	1	1111
011 ... 111	000 ... 111	0	$A + B$

The expression $d = \overline{a_2} \cdot \overline{a_0 \cdot a_1}$ corresponds to the three invalid entries above, that is, $a_2 = 0$ and $a_1 \cdot a_0 = 0$. In this case, d evaluates to 1, so Z produces all $1's$ regardless of the value of A or B. On the other hand, the expression $\overline{d} = a_2 + a_0 \cdot a_1$ corresponds to the remaining entries, that is, $a_2 = 1$ or $a_1 \cdot a_0 = 1$. In this case, d evaluates to 0, and Z performs the addition between A and B.

The above analysis of the unoptimized circuit in Figure 5.2 shows that it is possible to verify an arithmetic circuit under arithmetic constraints if the constraint-free part can be identified. The constraint-free part can be easily verified using algebraic

rewriting, starting with the output signature $8z'_3 + 4z'_2 + 2z'_1 + z'_0$. This signature is rewritten using the equations of the logic components until it reaches the primary inputs. At this point, the computed input signature $4a_2 + 2a_1 + a_0 + 4b_2 + 2b_1 + b_0$ is the proof that the original (black) portion of the circuit is performing a 3-bit addition.

However in a real optimized design, a separation between the original and the modified circuit is unlikely. Now, let us assume that the circuit in Figure 5.2 is given without knowing the boundary between the black (original) and the red (added) portions of the circuit. The task is to explore the function of the entire circuit and to verify that it is a conditional circuit satisfying the condition $A \geq 3$. To do that, we can apply the standard procedure of algebraic rewriting, and starting with the output signature $8z_3 + 4z_2 + 2z_1 + z_0$ rewrite it in a reverse topological order, until reaching the primary inputs. The resulting polynomial is:

$$\begin{aligned} Sig = &-12a_0 \cdot a_1 - 11a_2 - 4a_0 \cdot a_1 \cdot a_2 \cdot b_2 - 2a_0 \cdot a_1 \cdot a_2 \cdot b_1 \\ &- a_0 \cdot a_1 \cdot a_2 \cdot b_0 + a_0 \cdot a_1 \cdot b_0 + a_0 \cdot a_2 + a_2 \cdot b_0 + 2a_0 \cdot a_1 \cdot b_1 \\ &+ 2a_1 \cdot a_2 + 2a_2 \cdot b_1 + 4a_0 \cdot a_1 \cdot b_2 + 4a_2 \cdot b_2 + 12a_0 \cdot a_1 \cdot a_2 + 15 \end{aligned}$$

from which it is hard to tell what the actual function is.

In order to verify that this is a conditional 3-bit adder $A + B$, satisfying the constraint $A \geq 3$, we can compare the computed signature Sig, although complex, with the expected signature provided by the designer. However, this is neither practical nor reliable. For a large circuit, the final signature may contains million of terms, since it is not a standard circuit with a linear signature anymore. As shown in the Results section, computing the signature for a 32-bit constrained multiplier will quickly run out of 16 GB memory. Even if the result can be achieved and compared with an expected signature, the process is not very reliable, as we cannot identify the function by analyzing such a complex polynomial.

At this point, the algebraic approach has lost its principle advantage, namely, the function of the circuit can be easily recognized from the computed signature and the specification can be written as a polynomial in a simple way. In the next section, the concept of *vanishing monomials* will be introduced, which can be used to solve this problem. We will then propose a case-splitting method to analyze and simplify the signature to make it work for general case.

5.3 Verifying Constrained Circuit by Case-splitting Analysis with Vanishing Monomials

5.3.1 Vanishing Monomials

From the analysis presented in the previous section, we conclude that we cannot obtain signature $(4a_2 + 2a_1 + a_0 + 4b_2 + 2b_1 + b_0)$ for the circuit in Figure 5.2 since the circuit correctly performs addition only under certain condition. We also showed that the condition is related to the terms a_2 and $a_0 \cdot a_1$. Hence, let us try to analyze the computed signature by partitioning their terms into two groups: the terms with a_2 or $a_0 \cdot a_1$, and the terms without a_2 and $a_0 \cdot a_1$. With this we obtain,

$$\begin{aligned} Sig = a_2(&-11 - 4a_0 \cdot a_1 \cdot b_2 - 2a_0 \cdot a_1 \cdot b_1 - a_0 \cdot a_1 \cdot b_0 + a_0 + b_0 + 2a_1 + 2b_1 \\ &+ 4b_2 + 12a_0 \cdot a_1) \;+\; a_0 \cdot a_1(-12 + b_0 + 2b_1 + 4b_2) \;+\; 15 \end{aligned} \tag{5.2}$$

We observe that the constant 15 is the only term that does not contain terms a_2 and $a_0 \cdot a_1$. Therefore, if both a_2 and $a_0 \cdot a_1$ are 0, then $Z = 8z_3 + 4z_2 + 2z_1 + z_0 = 15$. This means that $z_3 = z_2 = z_1 = z_0 = 1$, since they are Boolean variables. This agrees with the first three invalid entries of Table 5.1 in the previous section. Therefore, we can infer that when the circuit is in an invalid state ($A < 3$), the monomials a_2 and $a_0 \cdot a_1$ always evaluate to zero.

In our verification work, the monomials associated with invalid entries, in this case a_2 and $a_0 \cdot a_1$, are defined as ***vanishing monomials***. If the circuit is subjected to

certain arithmetic constraints, those conditions will appear as vanishing monomials in the final signature. The vanishing monomials help us simplify and analyze the signature generated in the verification process.

Going back to our analysis, when a_2 and $a_0 \cdot a_1$ vanish (evaluate to 0), the first three entries of each table in Figure 5.3 are selected, as shown in Figure 5.3(a), denoted Case1. In this case, all the output bits $\{z_0, z_1, z_2, z_3\}$ produce 1, indicating the *don't care* status. The opposite case is to select the remaining entries, as shown in Figure 5.3(b), Case2. This case can be further split into two cases: (1) Case3 in Figure 5.3(c); and (2) {Case4 in Figure 5.3(d) or Case5 in Figure 5.3(e)}. In theory, choosing Case4 or Case5 gives the same result, as they cover all the possible assignments. We will show, however, that choosing Case5 is more efficient.

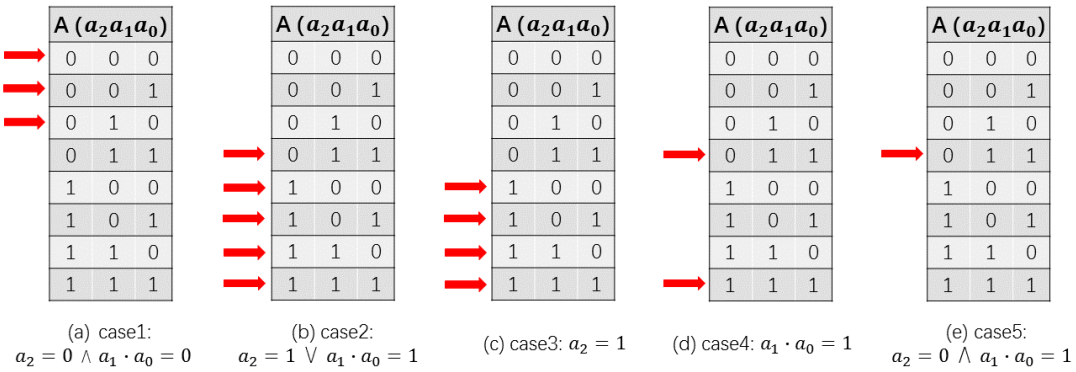


Figure 5.3: Different cases of entry selection.

To select the four entries in Figure 5.3(c), we set $a_2 = 1$. With this, the signature in Equation 5.2 becomes $Sig = a_0 + b_0 + 2a_1 + 2b_1 + 4 + 4b_2$. At the same time, the specification of the 3-bit adder $F_{spec} = a_0 + b_0 + 2a_1 + 2b_1 + 4a_2 + 4b_2$ becomes $a_0 + b_0 + 2a_1 + 2b_1 + 4 + 4b_2$. Therefore, $Sig = F_{spec}$, proving that when $a_2 = 1$, the circuit performs a 3-bit addition.

Similarly, to select the entry in Figure 5.3(e), we apply $a_2 = 0$ and $a_0 \cdot a_1 = 1$ to both Sig and F_{spec}. Notice that, since a_0 and a_1 are Boolean, $a_0 \cdot a_1 = 1$ implies

$a_0 = a_1 = 1$. In this case, $Sig = F_{spec} = b_0 + 2b_1 + 3 + 4b_2$, proving that the circuit performs 3-bit addition when selecting Case5. Finally, since all the entries, either valid or invalid, are covered, we can conclude that the circuit is indeed a 3-bit adder $Z = A + B$, under the condition $A \geq 3$.

5.3.2 Case-splitting Verification Approach

In this section, we provide another example to further explain the proposed case-splitting approach. In the previous sections, we discussed the scenario when constraint $X \geq c$ is applied to the circuit. The following example in Figure 5.4 illustrates a conditional 3-bit adder with a constraint $A < 3$, opposite to the one in Figure 5.2.

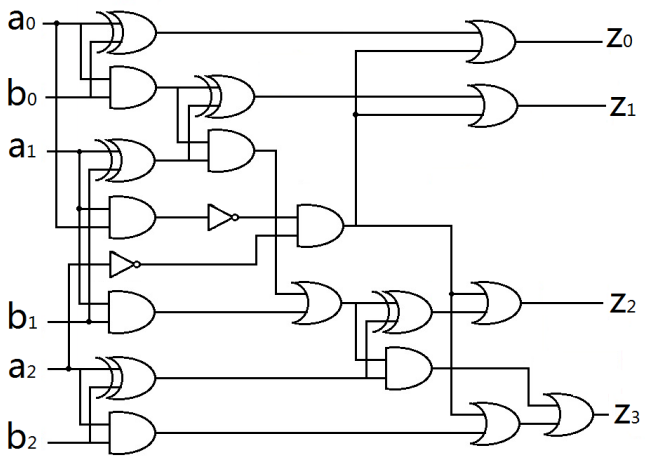


Figure 5.4: 3-bit adder $Z = A + B$, for $A < 3$.

In this example, the circuit in Figure 5.4 has been optimized. For a large design, it is unlikely to separate the constraint-free part from the entire circuit after optimization. Since the condition of this circuit is the complement of the previous one ($A \geq 3$), all the previous invalid states (the first three entries of Table 5.1) now become the valid states, and all the previous valid states become invalid.

After rewriting, the circuit's signature is shown in Eqn.(5.3) below.

$$Sig = a_2(-12a_0 \cdot a_1 - 4b_2 - 2a_1 - 2b_1 - a_0 - b_0 + a_0 \cdot a_1 \cdot b_0 + 2a_0 \cdot a_1 \cdot b_1 + 4a_0 \cdot a_1 \cdot b_2 + 15) \ + \ a_0 \cdot a_1(-4b_2 - 2b_1 - b_0 + 12) \ + \ (a_0 + b_0 + 2a_1 + 2b_1 + 4b_2) \qquad (5.3)$$

It is obvioulsy different than Eqn.(5.2), since it represents a different circuit. However, since the boundary between the valid and invalid states in the truth table is the same, the vanishing monomials are still a_2 and $a_0 \cdot a_1$. Also, because the vanishing monomials are the same, the case-splitting analysis of Sig is the same, composed of three cases: Case1, Case3, and Case5 in Figure 5.3.

To select Case1, Figure 5.3(a), we apply $a_2 = 0$ and $a_0 \cdot a_1 = 0$ to both Sig and the expected F_{spec} of the adder, and the two turn out to be equivalent. In this case, terms a_2 and $a_0 \cdot a_1$ always evaluate to zero in the functionally correct implementation. However, we cannot just assume that the circuit is correct; considering the valid entries only is not sufficient since it does not cover all the possible assignments. Therefore, we also need to select Case2 in Figure 5.3(b) in order to identify the invalid entries.

Similarly, Case2 be further split into two cases, Case3 in Figure 5.3(c) and Case5 in Figure 5.3(e). After applying $a_2 = 1$ or $\{a_2 = 0,\ a_0 \cdot a_1 = 1\}$ to Sig, we obtain $Sig = 15$, which agrees with the expected signature for invalid states. This is because only when $z_3 = z_2 = z_1 = z_0 = 1$, do we have $Z = 8z_3 + 4z_2 + 2z_1 + z_0 = 15$. Therefore, by case-splitting, we prove that the circuit in Figure 5.4 is a 3-bit adder $Z = A + B$, under the condition $A < 3$. As we can see, the case-splitting strategy for the verification of the two conditional adders (with $A \geq 3$ and $A < 3$) is the same, even though they have different signatures. Consequently, we can infer that the vanishing monomials are related to the constant c in the constraint $X \geq c$ or $X < c$. The method to systematically generate the vanishing monomials will be discussed in the next section.

The process of setting the vanishing monomials to either 0 or 1 is called *valuation of vanishing monomials*. The valuation of vanishing monomials help us choose between different cases. For efficiency reason, the valuation is not performed only after the final signature is computed. Instead, the valuation is applied during rewriting, once a term that contains one of the vanishing monomials is detected; in this case the term is simplified according to the pre-selected case. In this way, the redundant terms are eliminated as soon as possible in an attempt to avoid the intermediate polynomials growing dramatically. In the Result Section 5.4, a significant difference in the result between valuation and without valuation will be demonstrated. This is also the reason why we choose Case5 instead of Case4, since it provides a greater degree of simplification.

In summary, to verify a circuit subjected to an arithmetic constraint, we first generate the vanishing monomials using the knowledge of the given constraint. Then, a case-splitting approach is applied according to the generated vanishing monomials. The union of all cases will cover all possible input assignments. Each case requires an individual rewriting with different valuation of vanishing monomials. The circuit is correct, if the computed signature is equal to the specification in each case.

Specifically, there are two lists: 0-set and 1-set, each initialized to be empty. For each case, one of the vanishing monomials will be moved into the 1-set, so that it will be evaluated to 1 during the rewriting. After the current case is checked ($Sig = F_{spec}$), the corresponding vanishing monomial is moved from the 1-set to the 0-set, so that it will be evaluated to 0 in the next case. We repeat this procedure until the 0-set contains all the vanishing monomials, suggesting that they all have been successively evaluated to 1. The last case is to check when all the vanishing monomials vanish (evaluate to 0). If the relationship $Sig = F_{spec}$ is satisfied for all the cases, we conclude that the circuit is correct. The order in which the vanishing monomials are evaluated can be random, but a decreasing order (from MSB to LSB) is recommended for the

efficiency purpose. In the next section, we will discuss how to generate vanishing monomials and analyze the complexity for the case-splitting approach.

5.3.3 Generation of Vanishing Monomials

As explained in the previous sections, vanishing monomials play an important role in our verification approach. They are associated with invalid entries and reflect the arithmetic constraints in the final signature. In this section, we explain the reason for the appearance of vanishing monomials and describe the method for their detection. In the following theorem we refer to a truth table as the way to describe the function.

Theorem: *The input signature Sig_{in} of the circuit contains vanishing monomials associated with the dc-set of the truth table, regardless whether these don't-care entries are used during synthesis or not.*

Proof. Let F be a single output bit of an arithmetic function, corresponding to one of the output columns of the truth table. According to the standard design procedure, it can be implemented as a disjunction (OR) of product terms. Since product term is a conjunction (AND) of literals of individual variables, it is represented in an algebraic form also as a product of the corresponding variables in respective polarities. This is also true for products that include complemented variables, e.g., $a \wedge \neg b = a \cdot (1 - b) = a - a \cdot b$. Hence, any product from the valid entries of the truth table may contain vanishing monomials. The same argument applies to the case when input variables appear in different product terms; a disjunction of those terms will also create a product of the respective literals, according to the equation: $a \vee b = a + b - a \cdot b$, where a, b can be any product term. As a result, the signature expression generated during rewriting may contain product of variables that correspond to vanishing monomials. □

Our previous work [4] described an approach to generate vanishing monomials for constant dividers by computing algebraic expressions for the invalid entries. However,

such a method is inefficient, since applying algebraic rewriting to a single output bit misses the chance for polynomial cancellations. This causes an exponential increase in the size of intermediate polynomials, resulting in a memory problem. For the case of contiguous entries (block of entries with the same value of F) in the truth table, we come up with a solution without performing rewriting. For the case of non-contiguous entries, we might still rely the rewriting.

Based on the discussion in the previous section, we observe that the constraint $X \geq c$ or $X < c$ have the same vanishing monomials. Thus, we can infer that the vanishing monomials are related to the constant c in the constraint. The following examples illustrate the process of generating vanishing monomials.

Example1: Let an 8-bit operand $A \geq 201$. We first convert the decimal number 201 into binary 11001001, then fill its truth table, shown in Table 5.2.

Table 5.2: Truth table for function $F : A \geq 201$, where A is an 8-bit operand. The symbol "-" represents "don't care"

$A(a_7a_6a_5a_4a_3a_2a_1a_0)$	F
00000000 ⋮ 11001000	0
11001001	1
1100101-	1
110011--	1
1101----	1
111-----	1

According to the entry selection discussed in the previous section, if we want to select the last row the table, we set $a_7 \cdot a_6 \cdot a_5 = 1$. Then, $a_7 \cdot a_6 \cdot a_5 = 0$ represents the rest of the rows. According to this, in order to select the second row from the bottom, we can first set $a_7 \cdot a_6 \cdot a_5 = 0$, then $a_7 \cdot a_6 \cdot a_4 = 1$. Since the expression $a_7 \cdot a_6 \cdot a_5 = 0$ implies that a_7 or a_6 or a_5 is 0, and $a_7 \cdot a_6 \cdot a_4 = 1$ implies $a_7 = a_6 = a_4 = 1$, the combination of these two steps imply $a_7 = a_6 = a_4 = 1$ and $a_5 = 0$, allowing us to

select the second row from the bottom. Similarly, to select the third row, we can set both $a_7 \cdot a_6 \cdot a_5$ and $a_7 \cdot a_6 \cdot a_4$ to 0 and $a_7 \cdot a_6 \cdot a_3 \cdot a_2$ to 1.

Such a procedure is consistent with the 0-set and 1-set in the case-splitting strategy, discussed in the previous section. Therefore, we can infer that the monomials $a_7 \cdot a_6 \cdot a_5$, $a_7 \cdot a_6 \cdot a_4$ and $a_7 \cdot a_6 \cdot a_3 \cdot a_2$ are actually vanishing monomials. They simply agree with the index of $1's$ in each row. After collecting all the vanishing monomials and performing factorization, we realize that we can directly get the expression of vanishing monomials from the binary number $11001001 = (201_{10})$. The following algorithm explains the procedure.

Scanning the binary string from the MSB to the LSB, we replace each binary number 1 by a string "$a_i \cdot ($" and replace 0 by "$a_i +$". The character ")" is then added in the end. Thus, according to this procedure, 11001001 is replaced by:

$$\begin{aligned} & a_7 \cdot (a_6 \cdot (a_5 + a_4 + a_3 \cdot (a_2 + a_1 + a_0 \cdot 1))) \\ & = a_7 \cdot a_6 \cdot (a_5 + a_4 + a_3 \cdot (a_2 + a_1 + a_0)) \\ & = a_7 \cdot a_6 \cdot a_5 + a_7 \cdot a_6 \cdot a_4 + a_7 \cdot a_6 \cdot a_3 \cdot a_2 + a_7 \cdot a_6 \cdot a_3 \cdot a_1 + a_7 \cdot a_6 \cdot a_3 \cdot a_0 \end{aligned} \tag{5.4}$$

The vanishing monomials are clearly identified, since they are separated by a plus sign +. When all the vanishing monomials evaluate to 0 (are added into the 0-set), the first row for $F = 0$ in Table 5.2 is selected.

Example2: Let $A < 108$, where A is an 8-bit operand. As before, we first convert the decimal number 108 into binary 01101100, and fill its truth table, Table 5.3.

In the same way, starting from the MSB to the LSB of binary number 01101100 (108_{10}), we replace each 1 by a string "$a_i \cdot ($" and replace 0 by "$a_i +$". The character ")" is finally added. According to this procedure, 01101100 is replaced by:

Table 5.3: Truth table of F:$A < 108$, where A is an 8-bit operand.
The symbol "-" represents "don't care"

$A(a_7a_6a_5a_4a_3a_2a_1a_0)$	F
00000000 ⋮ 01101011	1
01101100 01101101 0110111-	0
0111----	0
1-------	0

$$a_7 + a_6 \cdot (a_5 \cdot (a_4 + a_3 \cdot (a_2 \cdot (a_1 + a_0 + 1)))) = a_7 + a_6 \cdot a_5 \cdot (a_4 + a_3 \cdot a_2)$$

$$= a_7 + a_6 \cdot a_5 \cdot a_4 + a_6 \cdot a_5 \cdot a_3 \cdot a_2 \tag{5.5}$$

Therefore, there are three vanishing monomials a_7, $a_6 \cdot a_5 \cdot a_4$, and $a_6 \cdot a_5 \cdot a_3 \cdot a_2$. When these monomials all evaluate to 0, the first row for $F = 1$ in Table 5.3 is selected.

5.3.4 Complexity Analysis

The complexity of the case-splitting approach is related to the number of vanishing monomials. For m vanishing monomials, there are $m + 1$ cases to be checked. Specifically, each of the vanishing monomials is evaluated to 1, consecutively for each case. The last case is to put all the vanishing monomials into the 0-set to select the upper entries. Checking each case requires one run of the rewriting. Thus, there will be $m + 1$ times of algebraic rewriting needed with different valuation of vanishing monomials. However, all the cases can be evaluated in parallel, since the value of vanishing monomials in each case is known ahead of time. The total time is then dictated by the slowest case. Since each case performs the same rewriting steps (gate polynomials are the same), their performance would not vary much.

We now discuss the relationship between the size of the operand n (number of bits of the constant) and the number of vanishing monomials m. For the contiguous

entries (block of entries with the same value of F) in the table, the best case is $m = 1$, regardless of the value of n. This case occurs when valid entries and invalid entries cut the overall space in half-and-half, with the MSB being 0 or 1. As the boundary between the valid and invalid entries moves up or down, the value of m changes. According to this, the worst case is when there is only one valid entry, for example, when the constraint is $A > 0$, in which case $m = n$.

In the previous 3-bit adder example, the simulation would need $2^6 = 64$ test vectors to cover all possible assignments. In contrast, the case-splitting method has only two vanishing monomials, thus checking only three cases will cover all the possible assignments. In general, for a two n-bit operand circuit, simulation requires 2^{2n} test vectors, while the proposed case-splitting method has $n + 1$ cases to check, in the worst case. For both operands having constraints, there are n^2 cases to check in the worst case.

5.4 Results and Conclusion

The case-splitting approach and the modified algebraic rewriting technique have been implemented as a program in C++. The experiments were performed on an Intel Core CPU i5-3470, 3.20 GHz with 15.6 GB memory. The proposed method has been tested on the constrained arithmetic circuits: adders, multipliers and dividers, up to 128 bits. Standard arithmetic circuits are generated by the ABC tool [50]. The arithmetic constraints were manually imposed on the circuits and the circuit was resynthesized.

Table 5.4 shows the verification time of different approaches, namely SAT (Lingeling [9]), original algebraic rewriting [86], and the proposed case-splitting method. It turns out that building a miter between two circuits and trying to solve the SAT problem is not efficient as it requires bit-blasting, i.e. representing the circuit as a bit-level netlist. This approach could not verify constrained circuits beyond 16-bit

Table 5.4: Verification time of different approaches.

*$T.O$: Time out of 3,600 sec, $M.O$: Memory out of 10GB.

Operand size	Verification time for constrained adders (sec)			Verification time for constrained multipliers (sec)		
	SAT [9] (Lingeling)	Algebraic rewriting [86]	This work	SAT [9] (Lingeling)	Algebraic rewriting [86]	This work
8	1.01	0.01	0.01	3.54	0.03	0.01
16	123.2	6.10	0.01	T.O	8.24	0.02
32	T.O	823.4	0.03	T.O	M.O	0.13
64	T.O	M.O	0.09	T.O	M.O	2.10
128	T.O	M.O	0.72	T.O	M.O	9.94

operands. It should be noted that the case-splitting analysis can also be handled by SAT solvers, since the valuation of VMs can be achieved by simply adding clauses to a SAT formula. One observation is that, for the cases of invalid input assignment, SAT solver is more efficient than rewriting and quickly gives an unSAT answer, suggesting that the circuits are functionally equivalent. This is because, for invalid assignment, the conditional circuit outputs all '1', bypassing the major logic. However, for the valid cases, where the circuit is actually performing the required arithmetic operation, the SAT solver needs to examine a larger search space and is much slower than rewriting. In summary, SAT does not offer much benefit in case-splitting approach and shows a low scalability.

Direct application of algebraic rewriting to the constrained circuits is also not feasible. As discussed in Section 5.3.2, the unresolved vanishing monomials will cause the memory issue. In general, the constrained circuit has the form $F' = F \vee d$, where F represents the constraint-free part that can be quickly verified alone, and d is the constraint condition which can be represented by the vanishing monomials. The expression for the constrained circuit $F' = F \vee d$ can be given in an algebraic form as $F + d - F \cdot d$, where F represents the output signature of the constraint-free circuit, and polynomial d represents the imposed constraints condition. The rewriting of F is fast as explained in Chapter 3. However, the size of intermediate polynomial d and

$F \cdot d$ grows exponentially without polynomial simplification during the rewriting. As we can see in the table, calculating the signature for the constrained circuits beyond 32-bit use all the 10 GB memory (but still inconclusive).

In this work, we modify the algebraic rewriting technique to handle constraints by valuation of vanishing monomials. During the rewriting process, once we detect a term that contains one of the vanishing monomials, we simplify this term by valuation of VMs according to the pre-selected case. By doing this, we resolve the expression of d as soon as possible to avoid the size of intermediate signature becoming too large. Hence, each rewriting does not suffer from memory issue. This is the reason that the proposed method is fast and scalable, as shown in Table 5.4. The verification time reported in the table is the time to verify a single case. However, as explained earlier, since all of the cases can be checked in parallel, the total time is dictated by the slowest case. Moreover, since each case performs the same rewriting steps with different valuation of vanishing monomials, the performance does not vary much.

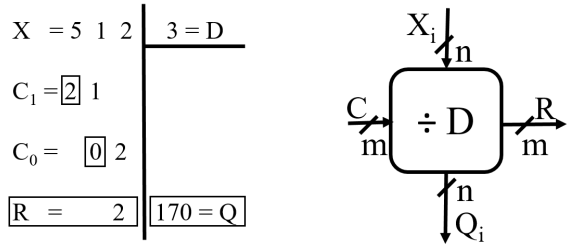
One might argue that if the applied constraint is changed, the verification time in Table 5.4 may change dramatically. To verify this opinion, we performed an experiment with a 64-bit multiplier to test if different constraints (with a different number of vanishing monomials) affect the verification time and to what extent. Different constraints were applied to the multiplier to affect the number of vanishing monomials. The results of this experiment are shown in Table 5.5.

Table 5.5: Verification time of a 64-bit multiplier ($A \times B$) with different constraints.

Arithmetic constraint	# Vanishing monomials	Verification time (sec)
$A \geq$ 0x7FFFFFFFFFFFFFFF	1	2.01
$A <$ 0x7FFF7FFFFFFBFFDF	4	1.99
$A \geq$ 0xF0FFFFFF9FFFFFF3	8	2.03
$A <$ 0xFFF0FFF0FF0FF0FF	16	2.11
$A \geq$ 0x00F90F0FF00F73F1	32	2.52
$A >$ 0x0000000000000000	64	2.50

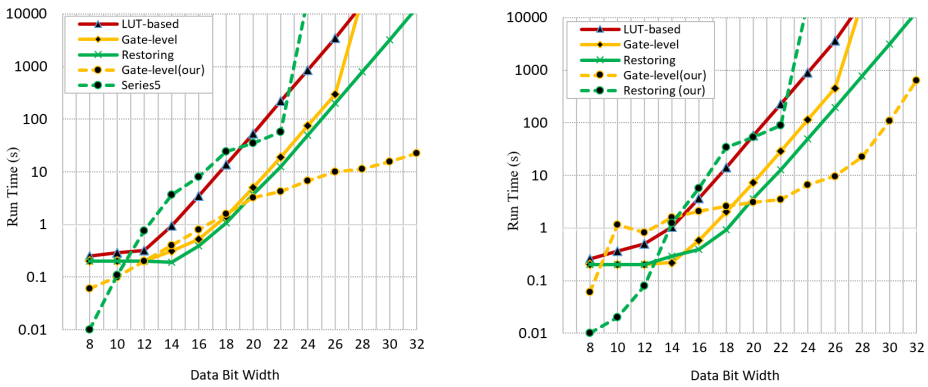
One observation from the table is that, as the number of vanishing monomials increases, the verification time does not increase much. This is because, when valuating the vanishing monomials, we avoid the increase in memory by resolving the algebraic expressions of d before they grow too much. Thus, an increase in the number of vanishing monomials would not cause memory problem, despite the fact that the increase in the number of vanishing monomials requires more cases to be checked. This is not a problem, however, since all the cases can be checked in a parallel. The only thing that would be affected is the search space in each step of the rewriting. During the rewriting, we need to check if a new term contains one of the vanishing monomials. Therefore, increasing the number of vanishing monomials require only a little more time for each substitution. Since this will not cause the memory overload issue, it has low net impact on the verification time.

The proposed case-splitting method has been also tested on a number of 32-bit constant dividers for different divisor values. The basic block of a constant divider is shown in Figure 5.5. Each block is subjected to a constraint $C < D$, where C is the input remainder, and D is the divisor value. We compared our method against an exhaustive simulation, the result of which is shown in Figure 5.6 and Table 5.6. Figure 5.6 shows the simulation results for $D = 257$ and 283 for the following three cases: 1) LUT-based implementation generated by FloPoCo [27]; 2) Gate-level implementation synthesized with ABC; and 3) A restoring constant divider implemented with ABC.



(a) Basic Division (b) Basic Block

Figure 5.5: Division operation and the basic divider block.



a Divisor = 257 b Divisor = 283

Figure 5.6: Exhaustive simulation run time for divisors D=257 and D=283 for different implementations, as a function of the dividend bit-width.

The results show that the simulation approach is competitive for dividend bit-widths up to 22 bits, but for higher bit-widths simulation becomes prohibitive. For example, the simulation for dividends larger than 28 bits required $15,264$ seconds (4h 24m), with memory-out at 24 GB for larger bit-widths.

Table 5.6: Verification results for the divide-by-constant divider circuit with a 32-bit dividend X using the proposed technique for Modular 1-bit block.

Divisor	# Rem. bit	Runtime w/o bug (sec)	# Bugs	Runtime w bugs (sec)
3	2	0.06	1	0.06
11	4	1.15	2	1.11
31	5	0.31	5	0.27
89	7	13.5	5	16.7
139	8	27.9	7	64.75
257	9	22.56	7	23.0
283	9	643.8	9	638.4

In the modular architecture, the boundary between adjacent blocks is known and the vanishing monomials are extracted and removed from the signature at each block, before rewriting the next block in series. The experiments in Table 5.6 include both correct (bug-free) and faulty circuits. The faults were emulated by randomly injecting multiple faults in the truth table into the valid portion of the look-up table. The result shows that this method can verify a 32-bit constant divider (one-bit block

architecture) for the divisor value up to 283. However, verifying non-modular divider architecture and a generic divider remains a challenging problem. The difficulty of verifying such dividers does not lie in the proposed case-splitting method, but in the divider itself (verifying constraint-free divider is hard). This problem requires more work and is a subject of a future research [5].

CHAPTER 6

VERIFICATION AND DEBUGGING OF GALOIS FIELD MULTIPLIERS

In this chapter, we present a novel method to verify and debug gate-level arithmetic circuits implemented in Galois Field arithmetic. The method is based on the computer algebra approach, similar to the one discussed in Chapter 3. However, instead of an algebraic backward rewriting, a forward reduction of the specification polynomials is applied to the circuit in $GF(2^m)$, using $GF(2)$ models of logic gates. To achieve this, we define a forward term order "$FO >$" and the rules of forward reduction that enable verification, bug detection, and automatic bug correction in the circuit. By analyzing the remainder generated by the forward reduction, the method can determine whether the circuit is buggy, and finds the location and the type of the bug. The experiments performed on Mastrovito and Montgomery multipliers show that our debugging method is independent of the location of the bug(s) and the debugging time is comparable to the time needed to verify the bug-free circuit.

6.1 Background

6.1.1 Galois Fields

A finite field or Galois field (GF) is a field that contains a finite number of elements. There are two main arithmetic operations in GF, addition and multiplication; other operations such as division can be derived from those two [57]. Galois field with p elements is denoted as GF(p). GF(p) is a *prime field* if it is consisted of a finite number of integers $\{1, 2, ..., p-1\}$, where p is a prime number, with additions and

multiplication performed *modulo p*. Of particular interest are *binary extension fields*, denoted GF(2^m) (or $\mathbb{F}_{2^m}$), which are finite fields with 2^m elements. The field of size m is constructed using *irreducible polynomial* $P(x)$, which includes terms of degree with $d \in [0, m]$ with coefficients in GF(2). Arithmetic operations in the field are then performed modulo P(x). GF arithmetic is used in hardware design for many cryptography applications, such as *Advanced Encryption Standard* (AES) and *Elliptic-curve cryptography* (ECC).

6.1.2 Computer Algebra Approach in GF

Algebraic approach to circuit verification used in this chapter relies on polynomial representation of the circuit specification and its logic gate components, discussed in Chapter 3. We recall here definitions of two signatures that play a major role in our algebraic-based verification. *Input Signature*, denoted Sig_{in}, is the polynomial in terms of the PI variables that represents the expected function of the circuit. *Output Signature*, denoted Sig_{out}, is a polynomial expressed in the PO variables that represents a binary encoding of the output. As defined in Chapter 3, algebraic backward rewriting is the process that transforms the output signature Sig_{out} into a unique input signature Sig_{in} using algebraic (polynomial) models of the logic gates of the circuit [22][87]. As discussed later in this chapter, the rewriting can also be done in the opposite direction (from PI to PO), referred to as *forward rewriting*.

This work models Boolean operators using algebraic models of $GF(2)$. For example, the pseudo-Boolean model of XOR in integer arithmetic, XOR(a, b)=$a + b - 2ab$, is reduced in GF(2) to $(a + b + 2ab) \bmod 2 = (a + b) \bmod 2$. The following algebraic models are used to describe basic logic gates in $GF(2^m)$ [46][87]:

$$\begin{aligned} \neg a &= 1 + a \bmod 2 \\ a \wedge b &= a \cdot b \bmod 2 \\ a \vee b &= a + b + a \cdot b \bmod 2 \\ a \oplus b &= a + b \bmod 2 \end{aligned} \tag{6.1}$$

To address the debugging problem of gate-level GF circuits we define two orders by which monomials are listed in each polynomial:

Definition 1: $BO >$ order: A backward, reverse-topological order, such that every output signal of a gate is always greater than its input signal. The set of polynomials representing a logic cone ordered according to $BO >$ is called a BO base.

Definition 2: $FO >$ order: A forward, topological order, such that every output signal of a gate is always less than its input signal. The set of polynomials ordered according to $FO >$ is called an $\underline{FO \text{ base}}$.

For example, the polynomials for an OR gate z in these orders are:

$BO >$ order: $z + a + b + a \cdot b$,

$FO >$ order: $a + b + a \cdot b + z$.

The $BO >$ order has been used successfully in verification works of [63][86][87], but, as demonstrated in this paper, it is ineffective in debugging of GF circuits. The approach described in this paper is based on an $FO >$ order.

6.1.3 GF Multiplier Principles

Galois Field multiplication is performed modulo an irreducible polynomial $P(x)$, a polynomial that cannot be factored into nontrivial polynomials over the given field [57][53]. The inputs and outputs of GF(2^k) multiplication are k-bit binary numbers. An example of a 4-bit GF(2^4) multiplication, with irreducible polynomial $P(x)$=x^4+x^3+1, is shown in Figure 6.1.

$$
\begin{array}{ccccccc}
 & & & a_3 & a_2 & a_1 & a_0 \\
 & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
s_6 & s_5 & s_4 & s_3 & s_2 & s_1 & s_0
\end{array}
$$

$$s_q = \bigoplus a_i b_j, \; \forall \; i{+}j{=}q, \; 0 \le q \le 6$$

s_3	s_2	s_1	s_0
s_4	0	0	s_4
s_5	0	s_5	s_5
s_6	s_6	s_6	s_6
z_3	z_2	z_1	z_0

$$
\begin{aligned}
z_0 &= s_0 \oplus s_4 \oplus s_5 \oplus s_6 \\
z_1 &= s_1 \oplus s_5 \oplus s_6 \\
z_2 &= s_2 \oplus s_6 \\
z_3 &= s_3 \oplus s_4 \oplus s_5 \oplus s_6
\end{aligned}
$$

Figure 6.1: Multiplication in GF(2^4): Z mod $P(x) = A \cdot B$ mod $P(x)$, where $P(x)$=x^4+x^3+1.

The GF multiplication is performed in a straightforward way by: 1) generating and adding the partial products; and 2) reducing the result over GF(2^m) with $P(x)$. The partial products are generated in the same way as in the integer multiplication, using AND operations. However the sum of the partial products (denoted s_q in Figure 6.1) is obtained using a series of XORs, since additions in finite field are implemented as XOR operations. This multiplication structure is called Mastrovito multiplier [76]. Note that in GF arithmetic there is no carry propagation between the columns of the result bits. Hence, each bit can be computed separately as a linear (XOR) sum of the product terms in the respective column. An alternative method for performing fast modular multiplication is the Montgomery multiplication [51]. It works by transforming two integer inputs, A, B, into Montgomery forms, $AR \; mod \; N$ and $BR \; mod \; N$, for some constant R, and computing the product $ABR \; mod \; N$. The multiplication result $A \cdot B$ is then obtained by transforming the result from the Montgomery form, as $A \cdot B = A \cdot B \cdot R^{-1} mod \; P(x)$. Since the Montgomery form generator is a GF multiplication with a constant input R, each of the four components can be considered as a simplified Mastrovito multiplication [42].

6.2 Bug Identification

This section describes an algorithm that utilizes the property of GF multipliers to identify bug(s) in the circuit. It consists of two major parts: 1) Remainder Generation and 2) Bug Analysis.

Algorithm 4 Remainder Generation via Forward Rewriting

Input: Gate-level netlist of the GF circuit
Input: Expected $InputSignature_i$ of each output bit
Output: Residual polynomials {$Remainder_i$}
Output: Non-zero elements of FO_i base

1: $\mathcal{PO}$=$\{z_0, z_1, ..., z_{n-1}\}$: primary output bits of the GF circuit
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: Extract cone PO_i from the gate-level netlist and generate the FO_i base
4: $Spec_i \leftarrow InputSignatrue_i - z_i$
5: **while** $Spec_i \neq 0$ **do**
6: $Success_{in} \leftarrow 0$
7: **for** each polynomial P_j in FO_i base **do**
8: **for** each monomial M_k in P_i **do**
9: **if** M_k divides $Spec_i$ **and** $Deg(M_k) == Deg(Spec_i)$ **and** M_k is not the last monomial in P_j **then**
10: Reduce $Spec_i$ by P_j; Set P_j in FO_i to 0; $Success_{in} \leftarrow 1$
11: **end if**
12: **end for**
13: **end for**
14: **if** $Success_{in} == 0$ **then**
15: Move the leading term of $Spec_i$ into $Remainder_i$
16: **end if**
17: **end while**
18: **end for**
19: **return** {$Remainder_i$, non-empty FO_i base}

Algorithm 4 describes forward rewriting; it imposes the $FO >$ order on the polynomials and reduces the specification by the FO base. Each polynomial representing a logic gate in the FO base has the form: $\{head\}, z_i$. The $\{head\}$ is the set of head monomials representing the gate inputs; the tail monomial z_i is the output of the gate. The degree of the monomial is the sum of the degrees of all its variables. For example, polynomial of an XOR gate is $a + b + z_i$, where the head set is $\{a, b\}$, each of degree one, and the tail is z_i.

Let the specification of the circuit be defined as $Sig_{in} - Sig_{out}$, the difference between the correct (expected) world-level input signature Sig_{in} and the output signature Sig_{out}, which is the binary encoded outputs. Since in a GF circuit the output bits are independent from each other, we can define $Spec_i$, the bit-level logic of output z_i, as $Sig_{in}(i) - z_i$. The input signature $Sig_{in}(i)$ is known, computed using the irreducible polynomial $P(x)$. The verification goal is then to reduce each specification $Spec_i$ by its FO_i base. If the result is 0, we conclude that the logic cone associated with bit i is bug-free. Otherwise, a non-zero $Remainder_i$ will indicate the presence of a bug (or bugs) in that cone.

The first step of Algorithm 4 is to extract all the cones from the circuit and derive the corresponding $Spec_i$ (lines 3-6). Recall that the FO_i base is the set of polynomials with an $FO >$ order describing the input-output relationship for cone i. While creating logic cones for each output, the common logic is duplicated, effectively making each cone $fanout\text{-}free$[1]. This means that every intermediate signal will appear only once in the head monomials of FO_i base.

The next step is to reduce each $Spec_i$ by its corresponding FO_i base. In lines 8-10, the Algorithm scans the polynomials in the FO_i base to check if the leading term lt_i of $Spec_i$ is divisible by *any* of the *head* monomials with the same degree as lt_i. This reduction is different than in the polynomial reduction based on the $BO >$ order, where only a *leading* monomial is used in the division process. Allowing the use of *any* of the head monomials is essential in identifying the bugs.

As an example, consider the reduction of $Spec_i = a + b + R$, where R is the remaining set of monomials, by polynomial $f = a + b + z$. The result of such a reduction should be $Spec_i = z + R$. However, if the monomial a is missing in $Spec_i$

[1]This is true for most structures such as Mastrovito multipliers, but may not be true for more complex ones, such as Montgomery; we currently limit our attention to those structures that can be made *reconvergent fanout free*

due to a bug (when the gate with output a is false), i.e., when $Spec_i = b + R$, the head monomial b in f still can be used to divide $Spec_i$. The result of such a reduction will be $Spec_i = a + z + R$. The monomial a in the reduced $Spec_i$ will never be eliminated by other polynomials in FO_i base, since there would be no other gate with signal a as input. When examining the content of the final remainder, it will be clear that the gate with output a is the source of the bug.

In summary, if there is a polynomial P_i in FO_i base that contains a head monomial that divides $Spec_i$, it will be used to reduce $Spec_i$. The polynomial P_i will then be removed from the base (each base polynomial can only be used once during the reduction). However, if the leading term of $Spec_i$ is not divisible by any of the head monomials in the FO_i base, it will be moved into $Remainder_i$. This process will be repeated until $Spec_i$ becomes empty (lines 13-18); or until it cannot be reduced anymore, in which case the content of the Remainder will be used to identify the bug.

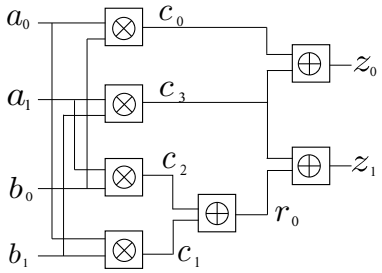


Figure 6.2: A two-bit Mastrovito GF multiplier.

Example 1: Consider a bug-free two-bit Mastrovito multiplier in Figure 6.2. The circuit can be separated into two cones: z_0 and z_1. The FO_1 base of cone z_1 includes polynomials: $p_1 = a_0b_1 + c_1$; $p_2 = a_1b_0 + c_2$; $p_3 = a_1b_1 + c_3$; $p_4 = c_1 + c_2 + r_0$ and $p_5 = c_3 + r_0 + z_1$, each written in $FO >$ order. According to the definition, $Spec_1$ is equal to the input signature of z_1 minus z_1, the output signature. The expected input signature of the circuit is $F = (a_0 + Xa_1) \cdot (b_0 + Xb_1) = a_0b_0 + X(a_0b_1 + a_1b_0) + X^2a_1b_1$. After $GF(2)$ reduction with the irreducible polynomial $P(X) = X^2 + X + 1$, we obtain $F = X(a_0b_1 + a_1b_0 + a_1b_1) + (a_0b_0 + a_1b_1)$. Hence, for output z_1, associated with X,

we have $Spec_1 = (a_0b_1 + a_1b_0 + a_1b_1) + z_1(mod\ 2)$. Similar expression can be derived for $Spec_0$ of output z_0.

The next step is to reduce the specification $Spec_1$ by the polynomial base of FO_1. This process is shown in Figure 6.3. We can see that $Spec_1$ is eventually reduced to 0. That is, $Remainder_1 = 0$ and all polynomials in FO_1 have been used, which indicates that cone z_1 is bug-free. We proceed similarly with bit z_0 to determine that its logic cone is also bug-free.

Example 2: Let us now consider the case when there is a *bug* in the two-bit multiplier in Figure 6.2. Let the bug be caused by replacing the XOR gate r_0 with an AND gate in cone z_1. That is, polynomial $p_4 = c_1+c_2+r_0$ is replaced by $p_4 = c_1c_2+r_0$ in the FO_1 base, while $Spec_1$ remains the same.

The process of reducing $Spec_1$ by the new FO_1 base is shown in Figure 6.3. Note that in the 4th iteration of the reduction, the polynomial $c_1 + c_2$ in $Spec_1$ is not divisible by any of the head monomials in FO_1, so it will be moved into $Remainder_1$. The monomial r_0 in the 5th iteration also be moved to $Remainder_1$.

Polynomials in FO_1 base of a bug-free cone	$Spec_1 = a_0b_1+a_1b_0+a_1b_1+z_1$
$p_1 = a_0b_1+c_1$	$Spec_1/p_1 = c_1+a_1b_0+a_1b_1+z_1$
$p_2 = a_1b_0+c_2$	$Spec_1/p_2 = c_1+c_2+a_1b_1+z_1$
$p_3 = a_1b_1+c_3$	$Spec_1/p_3 = c_1+c_2+c_3+z_1$
$p_4 = c_1+c_2+r_0$	$Spec_1/p_4 = 2c_2+c_3+r_0+z_1\ mod\ 2$
$p_5 = c_3+r_0+z_1$	$Spec_1/p_5 = 2r_0+2z_1 = 0\ mod\ 2$
Polynomials in FO_1 base of a buggy cone	$Spec_1 = a_0b_1+a_1b_0+a_1b_1+z_1$
$p_1 = a_0b_1+c_1$	$Spec_1/p_1 = c_1+a_1b_0+a_1b_1+z_1$
$p_2 = a_1b_0+c_2$	$Spec_1/p_2 = c_1+c_2+a_1b_1+z_1$
$p_3 = a_1b_1+c_3$	$Spec_1/p_3 = c_1+c_2+c_3+z_1$
$p_4 = c_1c_2+r_0$	$Spec_1/FO_1 = c_1+c_2+c_3+z_1$
$p_5 = c_3+r_0+z_1$	$Spec_1/p_5 = c_1+c_2+r_0+2z_1\ mod\ 2$

Figure 6.3: Generating *Remainder* with Forward Rewriting of bug-free and buggy logic cone of output bit z_1. $Remainder = 0$ for bug-free cone, and $Remainder = c_1 + c_2 + r_0$ for a buggy cone.

As a result, we obtain a non-zero $Remainder_1 = c_1 + c_2 + r_0$, and a non-empty FO_1 base: $p_4 = c_1c_2 + r_0$. This is a clear manifestation of a bug; namely, the XOR gate $(c_1 + c_2 + r_0)$ has been replaced by an AND gate $(c_1c_2 + r_0)$, while during the reduction, polynomial $c_1 + c_2$ should have been canceled by r_0. In a similar fashion we can identify other types of errors caused by gate replacement.

Table 6.1: Bug Analysis

Bug Type	Correct gate	False gate	Remainder	Non-empty base
1	XOR	AND	$a + b + z$	$a \cdot b + z$
2	XOR	OR	$a \cdot b$	-
3	AND	XOR	$a \cdot b + z$	$a + b + z$
4	AND	OR	$a + b$	-
5	OR	XOR	$a \cdot b$	-
6	OR	AND	$a + b$	-

Table 6.1 shows some common cases of erroneous gate-replacement in GF circuit for 2-input logic gates: AND, OR, and XOR. Similar relations can be derived for other gates as needed. In the table, signals a and b represent the inputs, and z is the output of the false gate. By analyzing the $Remainder_i$ and the non-empty FO_i base generated by the algorithm, we can readily determine the type of the error and locate the bug. This is possible because the remainder contains the names of the input and output signals of the false gate.

The bug of type $2, 4, 5, 6$ are all associated with the OR gate. Since the polynomial of the OR gate is $f = a + b + ab + z$, the *Spec* polynomial can be reduced by f, regardless whether the inputs are in the sum form $(a + b)$ or the product form $(a{\cdot}b)$. This means that the forward reduction will always "go through" the bug, and leave the residual polynomial with their inputs $(a + b$ or $a \cdot b)$ in the remainder. As a result, they do not have *non-empty base* and the *Remainder* contains only the input variables of the false gate, making it easy to identify the bug. On the other hand, for bug types 1 and 3, both the input signals (a, b) and the output signal (z) of the

false gate appear in the *Remainder*. Furthermore, a *non-empty base* indicates that the polynomial propagating through the circuit during forward rewriting cannot "go through" the bug. As we can see in the next section, multiple of these bugs will affect each other.

At this point the reader should fully appreciate the difference between forward rewriting (using $FO >$ order) and backward rewriting (using $BO >$ order). For a bug-free circuit both approaches will produce a zero remainder, indicating that the circuit implements the correct function. However, for the buggy circuit the remainders will be different. By construction, under the $BO >$ order, the remainder will contain only the primary inputs, but not the input and output signals of the false gates. This does not provide sufficient information about the type of the bug and its location. For example, assuming the same bug as in Example 2, the backward rewriting would produce $Rem_1 = a_0b_0a_1b_1 + a_1b_1$, with no indication as to the source of the bug. Furthermore, a single bug can make the size of remainder very large, making the analysis of the source of the bug difficult and the debugging process very hard. In contrast, in forward reduction, the remainder contains the signal name of the faulty gate and the location of the bug does not affect the size of the remainder.

6.3 Multiple Bugs Analysis

The debugging method using forward reduction described in the previous section can be extended to multiple bugs. In general, arithmetic bugs can be divided into *independent bugs* and *dependent bugs*. Independent bugs are those that will not affect each other; typically they appear in different cones. Since each cone is verified separately, each independent bug can be treated as a single bug in its own cone.

Dependent bugs can be classified into four cases, shown in Figure 6.4. The blank circle in the figure represents the correct gate. The shaded circle represents a *false gate*, i.e., the gate that was erroneously replaced by another gate. It can be of any type

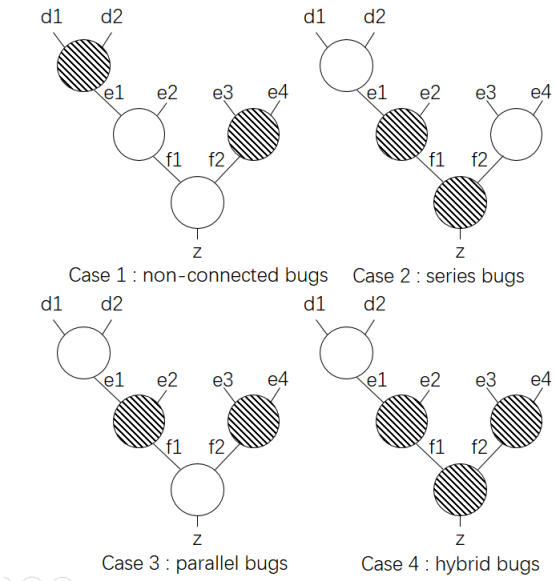


Figure 6.4: Different cases for dependent bugs.

discussed in Table 6.1, and even extended to other gates. Assume that in a correct implementation the gate e_1 is an AND gate and the remaining gates are XOR. With this, the specification of the cone with output z is $Spec = d_1d_2 + e_2 + e_3 + e_4 + z$.

The remainder in the case of multiple bugs depends not only on the cases shown in Figure 6.4, but also on the type of the bugs, listed in Table 6.1 (c.f. Section 6.2). Recall that intermediate input variables appear exactly once in each fanout-free cone. For this reason, the remainder for the circuit with bugs of type 2, 4, 5, 6 of Table I (i.e., those that have only false inputs left in the remainder) will be composed of disjoint sets of complete input pairs. As a result, the bugs of these types will not affect each other, regardless how they are connected (c.f. Case 1, 2, 3, 4 in Figure 6.4). One just needs to identify the complete input pairs in the remainder to detect the bugs. On the other hand, as mentioned in Section 6.2, for bug types 1 and 3 of Table I, both the input signals and the output signal of the false gate will appear in the *Remainder*. For this reason, when multiple bugs appear together, they will affect each other.

Example 3: Consider Case 1 in Figure 6.4, where the bugs at e_1 and f_2 are not connected directly. Assume that the AND gate e_1 is replaced by an XOR, and the XOR gate f_2 is replaced by an AND. The remainder computed by Algorithm 4, $Rem = d_1d_2 + e_1 + e_3 + e_4 + f_2$ can be partitioned into two disjoint groups, based on the input and output variables, such that each group forms a complete input-output pair: $(d_1d_2 + e_1)$ and $(e_3 + e_4 + f_2)$. In this case, it is always possible to achieve such a partition, since the two bugs are not connected directly. Specifically, the analysis of the non-zero base tells us that d_1, d_2, e_3, e_4 are the input signals (the head monomials), and e_1, f_2 are output signal (the tail monomials). Therefore, both bugs can be detected, one for e_1 gate and the other for f_2 gate.

Analysis of the other cases, where the false signals interact with each other, is more complex, as illustrated by the following example.

Example 4: Consider Case 2 in Figure 6.4, when the XOR gates f_1 and z are replaced by the AND gates. Here we cannot find a complete input-output pair in $Remainder = e_1 + e_2 + f_2 + z$, because signal f_1 is not only the false output of gate f_1 but also the false input of gate z. In this case we need to search for incomplete pairs in order to properly identify the bugs. For Cases 3 and 4, with the same $Remainder = e_1 + e_2 + e_3 + e_4 + z$, the input-output pairs cannot be found. The result depends on how z can be expressed as a combination of f_1, f_2 that partitions the remainder into groups. The detailed discussion of these cases is beyond the scope of this thesis proposal.

6.4 Results and Conclusions

The debugging technique described in this paper was implemented in Python and interfaced with the computer algebra tool, Singular [29], to affect the polynomial reduction. The experiments were performed on an Intel Core CPU i5-3470 @ 3.20 GHz 4 with 15.6 GB memory, using Mastrovito multipliers up to 256 bits as benchmarks

[76]. The more complex and challenging Montgomery multipliers [42] have only been partially tested and not reported here.

Table 6.2: Results of Mastrovito multipliers with single bug per cone.

Largest cone Operand size	# polys	Runtime for bug-free cone (sec)	Avg. runtime for buggy cone (sec)	Max runtime for buggy cone (sec)
z_5, 16-bit	167	0.36	0.36	0.37
z_{21}, 64-bit	539	6.2	6.3	6.3
z_{63},128-bit	1167	19.3	19.4	19.5
z_{10},256-bit	2033	46.3	46.5	46.6

Table 6.2 shows the verification results for a single bug inserted randomly in the circuit and illustrates the fact that the location of the bug does not affect the verification performance. To ensure that the location of the bug is the only variable factor in this experiment, for each circuit we extracted the largest output logic cone and inserted a single bug in it. The experiment was repeated for each cone 20 times, each time randomly changing the location of the bug, so they can be anywhere in the circuit and of any type shown in Table 6.1. The average time of the experiments was computed and the worst case runtime it took to locate the bug recorded. As we can see in the Table, the longest and average times are similar, which means that the time to locate and correct the bug does not depend on its location in the circuit. Furthermore, the time to verify the bug-free circuit is almost the same as the debugging of a single bug. In other approaches, the difference between these two times can be significant [46] [74].

Table 6.3 shows the debugging results for Mastrovito multipliers with multiple bugs. It gives the time it takes to verify the bug-free circuit, makes comparison with [62], and shows the number of bugs and the time to debug multiple bugs. The data of Table 6.3, in conjunction with that in Table 6.2, shows that the runtime for verifying the entire circuit is much less than the runtime of verifying a single cone multiplied

Table 6.3: Results of Mastrovito multipliers with multiple bugs.

Operand size	Runtime for bug-free circuit (sec)	Result of [62]	Number of bugs	Avg. runtime for multiple bugs circuit (sec)
8	0.33	0.09	16	0.37
16	0.84	0.42	24	0.92
32	3.89	0.83	32	4.23
64	30.39	28.90	40	31.30
128	283.72	924.3	48	286.94
163	667.38	3,546.0	56	676.07
256	2,111.43	6,728.0	64	2,135.92

by the number of cones. This is because the verification of each cone is performed in parallel since the cones are independent from each other.

Column 3 of Table 6.3 shows the performance of [62] that used backward reduction with the $BO >$ order and Gröebner basis. As we can see, our results are superior for circuit sizes above 64 bits. This suggests that $FO >$ order can be a better choice for GF verification. The major advantage of the $FO >$ order, however, is the performance of debugging, as shown in columns 2 and 5 of Table 6.3. We randomly inserted bugs (dependent or independent) in circuit. The runtime difference between the bug-free circuit and a buggy circuit is negligible. Even in the largest case of the 256-bit Mastrovito multiplier, with 64 bugs inserted, the runtime difference is insignificant.

In summary, our verification scheme based on forward reduction scheme offers an effective method for identifying and removing bugs in GF circuits. Unlike other methods, its performance does not depend on the location of the bug, and the time to locate the bug is comparable to verifying a bug-free circuit.

CHAPTER 7

CONTRIBUTIONS, PUBLICATIONS

Symbolic computer algebra approach is believed to be the most successful verification technique for arithmetic circuits verification [25]. Two flavors of these techniques dominate the field: one, based on Gröbner basis polynomial reduction [69][82][59][46][68][66] [65]; and the other, based on algebraic rewriting [21][86], described in this thesis.

Specifically, this work includes the following novel contributions:

- Formally analyzed the relation between algebraic rewriting and division-based GB reduction techniques, coming to a reasonable conclusion that the algebraic rewriting is more efficient from the implementation point of view.

- Contributed to the development of the *bit-flow model*, which provides an argument for soundness and completeness of the algebraic rewriting method, independently from the computer algebra arguments.

- Developed an efficient approach to the verification of *truncated multipliers*. To the best of our knowledge, it is the first successful approach to formally verify such circuits. The framework using re-synthesis technique can be applied to other truncated arithmetic circuits and other circuit verification problem.

- Proposed an efficient debugging method for Galois Field multipliers. To the best of our knowledge, it is the first computer algebra based debugging technique whose performance is not significantly affected by the bug location. In addition, this method does not suffer from memory overload problem.

- The *forward* variable order "$FO >$", proposed for the debugging method, enables performing verification and debugging at the same time. It also allows *forward rewriting* of GF circuits.

- Proposed a case-splitting method to efficiently verify arithmetic circuits subjected to arithmetic constraints. The concept of vanishing monomials is introduced to analyze the problem of arithmetic constraints.

Publications Related to this Work

- Atif Yasin, **Tiankai Su**, Sébastien Pillement, Maciej Ciesielski, "SPEAR: Hardware-based Implicit Rewriting for Square-root Verification" (to be appear), *Design, Automation and Test in Europe Conference* (DATE), 2020.

- Atif Yasin, **Tiankai Su**, Sébastien Pillement, Maciej Ciesielski, "Functional Verification of Hardware Dividersusing Algebraic Model", *IEEE international conference on Very Large Scale Integration (VLSI) and System-on-Chip (SoC) design* (VLSI-SOC), Oct. 2019.

- Maciej Ciesielski, **Tiankai Su**, Atif Yasin, Cunxi Yu, "Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model" , *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (TCAD 2019), April 2019.

- Atif Yasin, **Tiankai Su**, Sbastien Pillement, Maciej Ciesielski, "Formal Verification of Integer Dividers: Division by a Constant", *IEEE Computer Society Annual Symposium on VLSI* (ISVLSI'19), Miami, Florida, July 2019.

- Cunxi Yu, **Tiankai Su**, Atif Yasin and Maciej Ciesielski,"Spectral Approach to Verifying Non-linear Arithmetic Circuits", *ACM/IEEE Asia and South Pacific Design Automation Conference* (ASP-DAC'19), January 2019.

- Cunxi Yu, Atif Yasin, **Tiankai Su**, Alan Mishchenko and Maciej Ciesielski, "Rewriting Environment for Arithmetic Circuit Verification", *International Conference on Logic Programming and Automated Reasoning* (LPAR-22) vol 57, pages 656–666, November 2018.

- **Tiankai Su**, Atif Yasin, Cunxi Yu, Maciej Ciesielski, "Computer Algebraic Approach to Verification and Debugging of Galois Field Multipliers", *IEEE International Symposium on Circuits and System* (ISCAS'18) IEEE, May 2018.

- **Tiankai Su**, Cunxi Yu, Atif Yasin, and Maciej Ciesielski, "Formal Verification of Truncated Multipliers using Algebraic Approach", *IEEE Computer Society Annual Symposium on VLSI* (ISVLSI'17), Bochum, Germany, July 2017.

BIBLIOGRAPHY

[1] Electronic design automation consortium. In *About the EDA Industry* (Sep. 2014).

[2] Rand group. In *Engineering Change Order: A Simple Explanation For Why You Need One* (2014-10-09).

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

[4] Atif Yasin, Tiankai Su, Sébastien Pillement Maciej Ciesielski. Ieee computer society annual symposium on vlsi. In *Formal Verification of Integer Dividers:Division by a Constant* (July 2019).

[5] Atif Yasin, Tiankai Su, Sébastien Pillement Maciej Ciesielski. Ieee international conference on very large scale integration (vlsi) and system-on-chip (soc) design. In *Functional Verification of Hardware Dividersusing Algebraic Model* (Oct 2019).

[6] Balint, A., Belov, A., and Heule, M. Lingeling, Plingeling and Treengeling Entering the sat Competition 2013. *University of Helsinki B-2012-2* (2013), 51–52.

[7] Barr, Keith. New york: Mcgraw-hill. In *ASIC Design in the Silicon Sandbox: A Complete Guide to Building Mixed-signal Integrated Circuits* (July 2007), vol. 978-0-07-148161-8.

[8] Becker, Bernd, Drechsler, Rolf, and Werchner, Ralph. On the relation between bdds and fdds. *Inf. Comput. 123* (1995), 185–197.

[9] Biere, Armin. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT Competition* (2013), 51–52.

[10] Biere, Armin, Cimatti, Alessandro, Clarke, Edmund, and Zhu, Yunshan. *Symbolic model checking without BDDs*. Springer, 1999.

[11] Brayton, R., and Mishchenko, A. ABC: An Academic Industrial-Strength Verification Tool. In *Proc. Intl. Conf. on Computer-Aided Verification* (2010), pp. 24–40.

[12] Brock, Bishop, Kaufmann Matt, and Moore, J Strother. Formal methods in computer-aided design (fmcad). In *Acl2 theorems about commercial microprocessors* (1996), no. 275-293.

[13] Bryant, R. E., and Chen, Y-A. Verification of Arithmetic Functions with Binary Moment Diagrams. In *Design Automation Conference* (1995), pp. 535–541.

[14] Bryant, Randal E. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on 100*, 8 (1986), 677–691.

[15] Buchberger, B. *Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal*. PhD thesis, Univ. Innsbruck, 1965.

[16] Burch, Jerry R, Clarke, Edmund M, Long, David E, McMillan, Kenneth L, and Dill, David L. Symbolic model checking for sequential circuit verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 13*, 4 (1994), 401–424.

[17] Burgun Luc, Greiner Alain, and Eudes, Prado Lopes. Proceedings of the international conference on asic (asicon), pekin,. In *A Consistent Approach in Logic Synthesis for FPGA Architectures* (October 1994), vol. 104107.

[18] Chen, Jingchao. Minisat blbd. *SAT COMPETITION 2014* (2014), 45.

[19] Ciesielski, M., Kalla, P., Zeng, Z., and Rouzeyre, B. Taylor Expansion Diagrams: A Compact Canonical Representation with Applications to Symbolic Verification. In *Design Automation and Test in Europe, DATE-02* (2002), pp. 285–289.

[20] Ciesielski, M., Su, T., Yasin, A., and Yu, C. Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019), 1–1.

[21] Ciesielski, M., and W. Brown, A. Rossi. Arithmetic Bit-level Verification using Network Flow Model. In *Haifa Verification Conference, HVC'13* (Nov. 2013), Springer, LNCS 8244, pp. 327–343.

[22] Ciesielski, M, Yu, C, Brown, W, Liu, D, and Rossi, André. Verification of Gate-level Arithmetic Circuits by Function Extraction. In *Submitted to DAC 2015* (2015), ACM, pp. 1–6.

[23] Ciletti, Michael D. Prentice hall. In *Advanced Digital Design with Verilog HDL* (2010).

[24] Clarke, Edmund M, Grumberg, Orna, and Peled, Doron. *Model checking*. MIT press, 2000.

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

[26] Davis, Martin; Logemann, George; Loveland Donald. Communications of the acm. In *A Machine Program for Theorem Proving* (1962).

[27] De Dinechin, Florent, and Pasca, Bogdan. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers 28*, 4 (2011), 18–27.

[28] De Moura, Leonardo, and Bjrner, Nikolaj. Tools and algorithms for the construction and analysis of systems. In *Z3: An efficient smt solver* (2008).

[29] Decker, W., Greuel, G.-M., Pfister, G., and Schönemann, H. SINGULAR 3-1-6 A Computer Algebra System for Polynomial Computations. Tech. rep., 2012. http://www.singular.uni-kl.de.

[30] Desai, Kirti Sikri. Eda cafe. In *EDA Innovation through Merger and Acquisitions* (March 2010).

[31] Drane, Theo A., Rose, Thomas M., and Constantinides, George A. On the Systematic Creation of Faithfully Rounded Truncated Multipliers and Arrays. *IEEE Trans. on Computers 63*, 10 (Oct. 2014), 2513–2525.

[32] Drechsler, Rolf, and Große, Daniel. Verifying next generation electronic systems. In *Infocom Technologies and Unmanned Systems (Trends and Future Directions)(ICTUS), 2017 International Conference on* (2017), IEEE, pp. 6–10.

[33] Duggal, Vijay. Cad. mailmax pub. In *Cadd Primer: A General Guide to Computer Aided Design and Drafting-Cadd* (April 2000), no. 978-0962916595.

[34] Faugere, Jean-Charles. A New Efficient Algorithm for Computing Gröbner Bases (F4). *Journal of Pure and Applied Algebra 139*, 13 (1999), 61 – 88.

[35] Gao, Sicun. Counting zeros over finite fields with gröbner bases. *Master's thesis, Carnegie Mellon University* (2009).

[36] Gordon, Michael JC, and Melham, Tom F. Cambridge university press. In *Introduction to HOL A Theorem Proving Environment for Higher Order Logic* (1993).

[37] Hamaguchi, K., Morita, A., and Yajima, S. Efficient construction of Binary Moment Diagrams for verifying arithmetic circuits. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)* (Nov 1995), pp. 78–82.

[38] Hany, A., Ismail, A., Kamal, A., and Badran, M. 2013 saudi international electronics, communications and photonics conference. In *Approach for a unified functional verification flow* (April 2013), pp. 1–6.

[39] I. Grobelna, M. Grobelny, M. Adamski. Proceedings of the ninth international conference on dependability and complex systems depcos-relcomex, advances in intelligent systems and computing volume 286, springer international publishing switzerland. In *Model Checking of UML Activity Diagrams in Logic Controllers Design* (2014), no. 233-242.

[40] Kapur, D., and Subramaniam, M. Mechanical Verification of Adder Circuits using Rewrite Rule Laboratory. *Formal Methods in System Design 13*, 2 (1998), 127–158.

[41] Kaufmann, Matt, and Moore, J Strother. Compass96, systems integrity. software safety. process security. proceedings of the eleventh annual conference on computer assurance. In *Acl2: An industrial strength version of nqthm* (1996), no. 23-34.

[42] Koc, Cetin K, and Acar, Tolga. Montgomery multiplication in GF(2k). *Designs, Codes and Cryptography 14*, 1 (1998), 57–69.

[43] Kupferschmid, S., Becker, B., Teige, T., and Frnzle, M. Proof certificates and non-linear arithmetic constraints. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems* (April 2011), pp. 429–434.

[44] Lavagno, Martin, and Scheffer. Boston: Pearson/addison-wesley. In *Electronic Design Automation For Integrated Circuits Handbook* (2010), no. 0-8493-3096-3.

[45] Loveland, Donald W. Volume 6. north-holland publishing. In *Automated Theorem Proving: A Logical Basis. Fundamental Studies in Computer Science* (1978).

[46] Lv, J., Kalla, P., and Enescu, F. Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmatic Circuits. *TCAD 32*, 9 (September 2013), 1409–1420.

[47] Marques-Silva, J. P., and Sakallah, K. A. Grasp: A new search algorithm for satisfiability. *Proceedings of International Conference on Computer-Aided Design* (1996), 220–227.

[48] McCaffrey, James D. Software research, development, testing, and education. In *Validation vs. Verification* (April 2006).

[49] Mead, Carver A., and Conway, Lynn. Boston: Addison-wesley. In *Introduction to VLSI systems* (1980), no. 0-201-04358-0.

[50] Mishchenko, A, et al. Abc: A system for sequential synthesis and verification. *URL http://www. eecs. berkeley. edu/˜ alanmi/abc* (2007).

[51] Montgomery, Peter L. Modular multiplication without trial division. *Mathematics of Computation 44*, 170 (1985), 519–521.

[52] Moskewicz, Matthew W., Madigan, Conor F., Zhao, Ying, Zhang, Lintao, and Malik, Sharad. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001* (2001), pp. 530–535.

[53] Nagell, Trygve. *Introduction to Number Theory*. Almqvist & Wiksell Stockholm, 1951.

[54] Narayan, K. Lalit. New delhi: Prentice hall of india. In *Computer Aided Design and Manufacturing* (April 2008), no. 978-8120333420.

[55] Niemetz, Aina, Preiner, Mathias, and Biere, Armin. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation 9* (2015).

[56] Owre, Sam, Rushby John M, and Shankar, Natarajan. Automated deduction - cade-11. springer. In *PVS: A Prototype Ver- ification System* (1992), no. 748-752.

[57] Paar, Christof, and Pelzl, Jan. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[58] Parthasarathy, Ganapathy, Huang, Chung-Yang, and Cheng, Kwang-Ting. An analysis of atpg and sat algorithms for formal verification. In *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International* (2001), IEEE, pp. 177–182.

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

[60] Petra, Nicola, and Strollo, Antonio Giuseppe Maria. Design of Fixed-Width Multipliers with Linear Compensation Function. *IEEE Trans. Circuits Syst.I: Regular Papers 58*, 5 (May. 2011), 947–960.

[61] Pradhan, D., Abadir, M., and Varea, M. Recent advances in verification, equivalence checking amp; sat-solvers. In *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design* (Jan 2005), pp. 14–.

[62] Pruss, T., Kalla, P., and Enescu, F. Equivalence Verification of Large Galois Field Arithmetic Circuits using Word-Level Abstraction via Gröbner Bases. In *DAC'14* (2014), pp. 1–6.

[63] Pruss, Tim, Kalla, Priyank, and Enescu, Florian. 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 PP*, 99 (November 2015), 1.

[64] Raghavendra, M. Noormohammadpour; C. S. Ieee. In *Poster Abstract: Minimizing Flow Completion Times using Adaptive Routing over Inter-Datacenter Wide Area Networks* (2018).

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

[66] Ritirc, Daniela, Biere, Armin, and Kauers, Manuel. Column-wise verification of multipliers using computer algebra. In *FMCAD'17* (2017).

[67] S. Kirkpatrick, C. D. G. Jr., and Vecchi, M. P. Science, 220(4598):671680. In *Optimization by Simulated Annealing* (1983).

[68] Sayed-Ahmed, Amr, Große, Daniel, Kühne, Ulrich, Soeken, Mathias, and Drechsler, Rolf. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In *DATE'16* (2016), pp. 1–6.

[69] Shekhar, N., Kalla, P., and Enescu, F. Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing. *TCAD 26*, 7 (July 2007), 1320–1330.

[70] Sheldon B. Akers, Jr. Ieee transactions on computers. In *Binary Decision Diagrams* (June 1978).

[71] Smith, Michael John Sebastian. Addison-wesley professional. In *Application-Specific Integrated Circuits* (1997), vol. 978-0-201-50022-6.

[72] Sörensson, Niklas, and Eén, Niklas. MiniSat 2.1 and MiniSat++ 1.0 - SAT race 2008 editions. *SAT* (2009), 31.

[73] Stump, Aaron, Barrett, Clark W., and Dill, David L. CVC: A Cooperating Validity Checker. In *14th International Conference on Computer Aided Verification (CAV)* (2002), Ed Brinksma and Kim Guldstrand Larsen, Eds., vol. 2404 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 500–504. Copenhagen, Denmark.

[74] Su, Tiankai, Yu, Cunxi, Yasin, Atif, and Ciesielski, Maciej. Formal verification of truncated multipliers using algebraic approach and re-synthesis. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2017), pp. 415–420.

[75] Sullivan, Michael B., and Swartzlander, Earl E. Truncated Error Correction for Flexible Approximate Multiplication. *Signals, Systems and Computers (ASILOMAR) ACSSC.2012.6489023* (Nov 2012), 10.1109.

[76] Sunar, Berk, and Koç, Ç K. Mastrovito multiplier for all trinomials. *Computers, IEEE Transactions on 48*, 5 (1999), 522–527.

[77] Vahid, Frank. John wiley and sons. In *Digital Design with RTL Design, Verilog and VHDL (2nd ed.)* (2010), vol. 978-0-470-53108-2.

[78] Vasudevan, S., Viswanath, V., Sumners, R. W., and Abraham, J. A. Automatic Verification of Arithmetic Circuits in RTL using Stepwise Refinement of Term Rewriting Systems. *IEEE Trans. on Computers 56*, 10 (2007), 1401–1414.

[79] Vizel, Y.; Weissenbacher, G.; Malik S. Proceedings of the ieee. 103 (11). In *Boolean Satisfiability Solvers and Their Applications in Model Checking* (2015).

[80] Weste, Neil H. E. Harris, David M. Boston: Pearson/addison-wesley. In *CMOS VLSI Design: A Circuits and Systems Perspective* (2010), no. 978-0-321-54774-3.

[81] Whitesitt, J. Eldon. Courier corporation. In *Boolean Algebra and Its Applications* (24 May 2012).

[82] Wienand, O., Wedler, M., Stoffel, D., Kunz, W., and Greuel, G.-M. An Algebraic Approach for Proving Data Correctness in Arithmetic Data Paths. *CAV* (July 2008), 473–486.

[83] Wikipedia. Dpll algorithm. `https://en.wikipedia.org/wiki/DPLL_algorithm`. Last edited: 2019-04-09.

[84] Wikipedia. Semiconductor industry. `https://en.wikipedia.org/wiki/Semiconductor_industry`. Last edited: 2019-04-09.

[85] Wos, Larry; Overbeek, Ross; Lusk Ewing; Boyle Jim. Mcgrawhill. In *Automated Reasoning: Introduction and Applications* (1992), no. 23-34.

[86] Yu, Cunxi, Brown, Walter, Liu, Duo, Rossi, André, and Ciesielski, Maciej J. Formal verification of arithmetic circuits using function extraction. *TCAD 35*, 12 (2016), 2131–2142.

[87] Yu, Cunxi, and Ciesielski, Maciej J. Efficient parallel verification of galois field multipliers. *ASP-DAC'17* (2017).

[88] Yu, Cunxi, Ciesielski, Maciej J., and Mishchenko, Alan. Fast algebraic rewriting based on and-inverter graphs. *IEEE Trans. on CAD of Integrated Circuits and Systems 37*, 9 (2018), 1907–1911.

[89] Zwolinski, V. Litovski; Mark. Springer science business media. In *VLSI Circuit Simulation and Optimization* (Dec. 1996).