November 2017

Formal Analysis of Arithmetic Circuits using Computer Algebra - Verification, Abstraction and Reverse Engineering

CUNXI YU
ECE, University of Massachusetts Amherst

FORMAL ANALYSIS OF ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA

- VERIFICATION, ABSTRACTION AND REVERSE ENGINEERING

A Dissertation Presented

by

CUNXI YU

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

DOCTOR OF PHILOSOPHY

September 2017

Electrical and Computer Engineering

FORMAL ANALYSIS OF ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA

- VERIFICATION, ABSTRACTION AND REVERSE ENGINEERING

A Dissertation Presented

by

CUNXI YU

Approved as to style and content by:

Maciej Ciesielski, Chair

George S. Avrunin, Member

Daniel Holcomb, Member

Sandip Kundu, Member

Christopher V. Hollot, Department Head
Electrical and Computer Engineering

ABSTRACT

FORMAL ANALYSIS OF ARITHMETIC CIRCUITS USING COMPUTER ALGEBRA
- VERIFICATION, ABSTRACTION AND REVERSE ENGINEERING

SEPTEMBER 2017

CUNXI YU

B.Sc., ZHEJIANG UNIVERSITY CITY COLLEGE

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Maciej Ciesielski

Despite a considerable progress in verification and abstraction of random and control logic, advances in formal verification of arithmetic designs have been lagging. This can be attributed mostly to the difficulty in an efficient modeling of arithmetic circuits and datapaths 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. Approaches that rely on computer algebra and Satisfiability Modulo Theories (SMT) methods are either too abstract to handle the bit-level nature of arithmetic designs or require solving computationally expensive decision or satisfiability problems.

The work proposed in this thesis aims at overcoming the limitations of analyzing arithmetic circuits, specifically at the post-synthesized phase. It addresses the *verification*, *abstraction* and *reverse engineering* problems of arithmetic circuits at

an algebraic level, treating an arithmetic circuit and its specification as a properly constructed algebraic system. The proposed technique solves these problems by function extraction, i.e., by deriving arithmetic function computed by the circuit from its low-level circuit implementation using computer algebraic rewriting technique. The proposed techniques work on large integer arithmetic circuits and finite field arithmetic circuits, up to 512-bit wide containing millions of logic gates.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

With an almost unmanageable increase in the size and complexity of ICs and SoCs, hardware design analysis has become a dominating factor of the overall design flow [44]. Hardware verification is one of the essential procedures in the design flow that checks whether the actual hardware implementation has the correct specification. Specifically, verifying arithmetic computation units are particularly difficult due to the "bit blasting" issue, i.e., flattening the design specification into to a bit-level. The importance of arithmetic verification problem grows with an increased use of arithmetic modules in embedded systems to perform computation-intensive tasks for multimedia, signal processing, and cryptography applications.

Formal verification techniques can benefit greatly from abstractions of the functionality of the circuits being verified. Abstraction reduces the complexity of analysis of the design and may provide a hierarchical view of the *register transfer level* (RTL), which could be applied to system-level verification. Word-level abstraction specifically focuses on extracting a word-level representation of the function implemented by a gate-level design. For example, for an n-bit gate-level multiplier, the word-level function can be extracted as $Z = AB$. We can see that as the datapath size of the multiplier grows, the bit-level representation increases exponentially, while the word-level abstraction does not change. However, formal techniques for abstraction in gate-level design are challenging. The abstraction problems are even harder than the verification problem since there are no clear boundaries of the arithmetic functions in the design.

This thesis aims at overcoming the limitations of analyzing large arithmetic circuits, especially of the post-synthesized circuits. The approach proposed here addresses the *verification*, *abstraction* and *reverse engineering* problems in the algebraic domain, in which both the implementation and the specification of the arithmetic circuit are represented as pseudo-Boolean polynomials in respective variables (circuit signals). It solves the verification problem by extracting an arithmetic function computed by the circuit directly from its gate-level circuit implementation using *computer algebraic methods*, called *function extraction* or *algebraic rewriting*. This rewriting technique transforms the polynomial representing an encoding of the primary outputs (called the *output signature*) into a polynomial at the primary inputs (called the *input signature*). The computed arithmetic function can be used to verify the circuit against the given specification (i.e., the expected function of the design), or to decipher the function performed by the circuit. In the case of an incorrectly implemented function, this method will generate a counterexample (bug trace).

Regarding *abstraction*, a new canonical representation of arithmetic functions, called *algebraic spectrum*, is introduced in this thesis. *Algebraic spectrum* refers the coefficient distribution of the polynomial expressions that are computed using the rewriting technique. We prove that the coefficients distribution of arithmetic function is unique depending on the arithmetic operations. Finally, the verification and reverse engineering problems of Finite Field Arithmetic circuits are explored. For the finite field arithmetics, the main contributions include:

- 1) computer algebraic method is approved to be applied for parallel verification over GF(2^m).
- 2) the approach of analyzing the irreducible polynomials of finite field arithmetic is proposed.

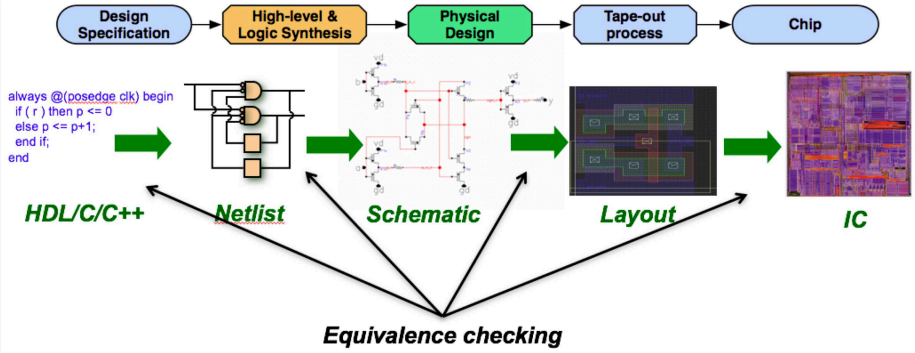


Figure 1.1: Typical industrial IC design flow.

- 3) the complete reverse engineering framework of finite field arithmetic is developed.

1.1 Hardware Verification

The importance and difficulty of arithmetic circuit verification can be illustrated by the famous $FDIV$ bug in Intel's Pentium processor in 1995, which cost Intel $475 million. This bug was not covered by the one trillion simulation vectors used for this processor [27]. Verification is a critical problem in the chip industry since the cost of hardware verification is claimed to be 70 percent of the overall hardware design effort. Although the engineers and researchers invest a lot in hardware verification, it is still challenging as the design complexity increases. The recent verification industry study presented by Foster [44] showed that the average verification time in the last ten years is around 60% (57% in 2014) over the entire chip design period. In this survey, it also shows that there are more than 50% designs requires more than 60% project time in verification only. Most of those designs contain large arithmetic units.

Hardware verification is a process of checking the correctness of the fabricated hardware compared to the specification. However, it is impossible to directly check if the fabricated hardware matches the original specification. Typically, hardware

verification is conducted step-by-step during the design flow. Hence, there are many hardware verification techniques developed that apply on different representations of hardwares, such as verification of HDL, gate-level netlist, schematic netlist, and so on. The design flow typically starts with a high-level specification using hardware description language (HDL) or C/C++. This specification is then compiled into a register-transfer-level (RTL) description, which is further optimized by high-level synthesis and logic synthesis techniques and translated into a corresponding netlist representation. The logic-level netlist is then translated to a physical layout during placement and routing synthesis.

Typically, equivalence checking has been applied at each step to check the equivalence before and after each optimization or transformation step in the design flow. There are many verification techniques that apply to different representations of hardware, such as HDL code, gate-level netlist, layout, etc. This thesis focuses on gate-level implementation of arithmetic circuits. Traditional approaches to verifying arithmetic circuits are based on simulation or emulation, but exhaustive simulation is not applicable to the large modern designs. Theorem proving approaches require verification experts to manually guide the systems to complete the proof. Thus, to automatically verify arithmetic circuits, many formal techniques have been developed to handle large practical circuits. However, few formal techniques are applicable to large gate-level arithmetic circuits. Those contemporary formal methods that could be applied to arithmetic circuits verification are reviewed in Chapter 2. The limitations of those formal methods for verifying arithmetic circuits are studied in the Chapter 3.

1.2 Verification Techniques

1.2.1 Equivalence Checking

In recent years, many CAD vendors have offered equivalence checking tools for design verification. Equivalence checking (EC) is one of the most widely used formal techniques in the verification of digital circuits. Depending on the type of the target circuits, i.e. sequential circuits or combinational circuits, equivalence checking can be classified in two types: *sequential equivalence checking* (SEC) and *combinational equivalence checking* (CEC).

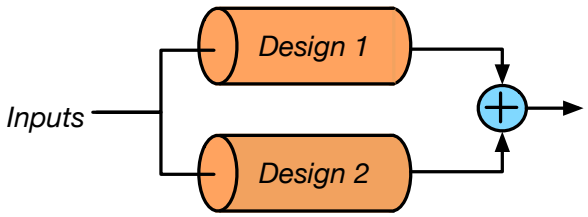


Figure 1.2: Combinational equivalence checking model.

The *Combinational Equivalence Checking* (CEC) model is shown in Figure 1.2. Let *Design1* be the design to be verified, and *Design2* be the reference design. The specification of the reference design is the expected specification of *Design1*. Given identical inputs to *Design1* and *Design2*, a *miter* is built by XORing the corresponding output bits of these two designs and connecting to a wide OR gate. If the functions represented by the two designs are identical for all input patterns, then *miter* always evaluates to 0 (Boolean false) for any input pattern. In this case, the two designs are proved to be equivalent. Otherwise, these two designs are not equivalent.

The *Sequential Equivalence Checking* (SEC) model is shown in Figure 1.3. SEC checks if the corresponding outputs are equivalent for any state of the two circuits with identical initial states. The proof of correctness of two sequential circuits requires a complete state-space traversal, which is one of the bottlenecks of sequential

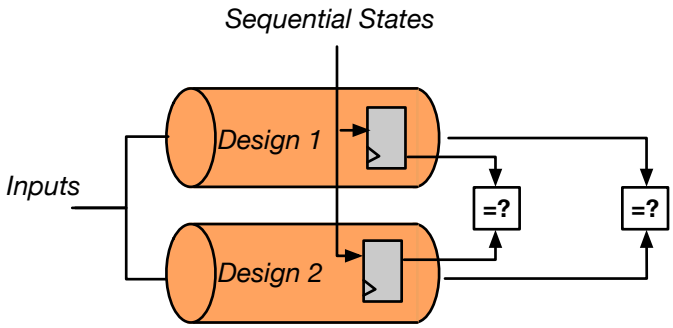


Figure 1.3: Sequential equivalence checking model.

equivalence checking. The complexity of sequential equivalence checking could be significantly increased by synthesis process, such as retiming technique [120].

The most straightforward technique to prove or disprove the equivalence using EC model is exhaustive simulation. It is obvious that exhaust simulation is not possible for design with a large number of input bits. And, random simulation does not provide complete proof of the equivalence because of the coverage problem. Further, formal verification methods are heavily investigated to address the equivalence checking problem. The most promising formal techniques for solving equivalence checking problem include *canonical diagrams*, *satisfiability* and *computer algebra methods*.

Reduced Ordered Binary Decision Diagrams (OBDDs) [20], provide the efficient method for equivalence checking for combinational and sequential circuits [5]. OBDDs are canonical representation with a fixed variable ordering. Hence, equivalence checking problem can be addressed by comparing the OBDDs of two designs. If two OBDDs are identical, two designs are functional equivalent. However, the size of the BDDs explodes for large designs. Specifically, it becomes large for the arithmetic circuits. For example, for integer multiplication, it has been shown that an n-bit multiplication requires a $O(n^3)$ size OBDD [24].

Due to the limitation of the BDDs, many techniques have been developed to reduce the complexity of equivalence checking. Goldberg et. al [47] presented a simple

framework for SAT-based CEC and reported results on an ISCAS-85 benchmarks. Paruthi et. al [88] proposed an idea that is based on a tight integration of a structural satisfiability (SAT) solver, BDD sweeping, and random simulation. In this work, the integral application of the SAT solver significantly enhances the capacity and efficiency of BDD sweeping and extends its suitability for mis-comparing designs. Further, the random simulation algorithm works on the graph that represents the netlist and thus runs more efficiently. Mishchenko et. al [76] presented an And-Inv-Graph (AIG) data structure that is the *state-of-the-art* technique for logic reduction and synthesis. The *Functional-reduced AIG* (FRAIG) is able to efficiently reduced the logic complexity [78] which has been implemented in the ABC system [80]. However, all these techniques are not applicable to large arithmetic circuits, such as Galois field multipliers and integer multipliers [31][72].

With significant research efforts spent on formal methods, formal verification for hardware that combines heuristic methods, such as checking structure similarity, becomes more and more popular. The similarity between the two circuits is exploited to identify the equivalences between internal nodes of the two circuits being checked for equivalence [63]. For example, the partial list of equivalence checkers are Formality (from Synopsys), Design Verifyer (from Chrysalis) and Verity (from IBM) [64]. Similarly, identifying structural similarities for sequential equivalence checking is also explored [120]. These tools perform logic equivalence checking of two circuits based structural analysis and BDD techniques. Similarly to other verification techniques, these equivalence checking techniques are limited by the memory explosion problem for arithmetic circuits. Recently, IBM Formal Verification team showed that their tools could automatically verify the floating point division (FDIV) unit using their SixthSense formal engine [62].

1.2.2 Model Checking

Model checking performs verification by exhaustively checking whether a state-transition graph (STGs) satisfies a given property [36]. In this approach, a circuit is described as a state machine with transitions to describe the circuit behavior. The specifications to be checked are described as properties that the machine should or should not satisfy. Model checking is limited by the *state-space explosion problem*. The state explosion problem refers to the fact that the number of states is exponential in the number of Boolean variables. Explicit state model checkers are based on graph-traversal of the model states, and must keep track of the visited states. However, this is infeasible due to the large size of the modern designs.

BDDs [20] have traditionally been used as a symbolic representation of the system. Model checkers based on BDDs are usually able to handle systems with hundreds of state variables. However, for larger systems, the BDDs generated during model checking become too large for currently available computers. Also, selecting the right ordering of BDD variables is very important. Boolean Satisfiability (SAT) also operates on Boolean expressions but does not use canonical forms. Biere et. al proposed a SAT procedures for symbolic model checking instead of BDDs [14][13]. They do not suffer from potential space explosion of BDDs and can handle propositional satisfiability problems with thousands of variables.

The symbolic model checking can be applied to sequential circuit verification [34]. However, due to the large size of arithmetic circuits, this technique is difficult to check the properties. Additionally, the symbolic model checking has been applied mostly to verifying the properties of systems instead of gate-level implementations.

1.2.3 Theorem Proving

Another class of solvers includes Theorem Provers, deductive systems for proving that an implementation satisfies the specification, using mathematical reasoning. The

proof system is based on a large and strongly problem-specific database of axioms and inference rules, such as simplification, rewriting, induction, etc. Some of the most popular theorem proving systems are: HOL [48], PVS [84], Boyer-Moore/ACL2 [19], and Nqthm [48][56]. These systems are characterized by high abstraction and powerful logic expressiveness, but they are highly interactive, require domain knowledge, extensive user guidance, and expertise for efficient use. The success of verification using theorem prover depends on the set of available axioms and rewrite rules, and on the choice and order in which the rules are applied during the proof process, with no guarantee for a conclusive answer. Similarly, term rewriting techniques, such as [121] or [55], are incomplete and "may fail to generate the proof because additional lemmas are needed" [55]. Additionally, theorem proving for gate-level circuits is extremely complex.

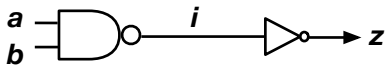


Figure 1.4: A gate-level circuit implementing 2-input AND function with inputs a and b.

For example, to prove that the circuit shown in Figure 1.4 is a AND2(a, b) function, theorem proving requires ten steps. The specification of z is F_{spec}=$a \wedge b$. The specification of the circuit is F=$\exists x.i\ NAND(a, b, x) \wedge NOT(x, z)$. The equivalence checking problem using theorem proving is: $\forall\ a, b, z\ F(a, b, z) \rightarrow F_{spec}$(a,b,z). The procedure of proving the equivalence between F_{spec} and F using theorem proving is shown in Figure 1.4.

1.2.4 Symbolic Simulation

Symbolic simulation is a well know technique for simulation and verification of digital circuits and is an extension of conventional digital simulation, where a simulator evaluates circuit behavior using symbolic Boolean variables to encode a range of cir-

$\vdash \exists$x. NAND(a,b,i) $\wedge$ NOT(i,z)
$\vdash$ NAND(a,b,i) $\wedge$ NOT(i,z)
$\vdash$ NAND(a,b,i)
$\vdash$ i = $\neg$ (a $\wedge$ b)
$\vdash$ NOT(i,z)
$\vdash$ z = $\neg$ i
$\vdash$ z = $\neg(\neg$(a $\wedge$ b))
$\vdash$ z = (a $\wedge$ b)
$\vdash$ AND(a,b,z)
$\vdash$ F(a,b,z) = F_{spec}(a,b,z)

Figure 1.5: Proving the function of output z in Figure 1.4 is AND(a,b) using theorem proving.

cuit operating conditions [21]. Several symbolic simulation tools were developed after Bryant introduced BDDs [18][9]. Bose et. al presented an automated verification tool of synchronous pipelined circuits, based on symbolic simulation [18]. The problem for this work and symbolic simulation is that the specifications must be expressed as Boolean functions, which can be very complex for some arithmetic circuits.

Compared with model checking, symbolic simulation technique can handle much larger circuits, because this approach can only cover some of the input spaces in each simulation run. However, symbolic simulation cannot be used to completely verify arithmetic circuits such as integer multipliers, field multipliers, etc., because the input spaces of these circuits are very large and the BDDs grow up exponentially for these circuits. An exhaustive simulation to cover the entire input space is almost impossible for large arithmetic circuits.

1.2.5 Reverse Engineering

Significant research effort is spent on reverse engineering for hardware security analysis and verification. Physical-based reverse engineering works can be done by decapsulating the chip using imaging and delayering techniques. Torrance and James show that reverse engineering of a fabricated integrated circuit chips can be very

successful [119]. Li et al. presented an automatic approach that identifies high-level components by matching a set of known components [68], and Subramanyan et al. improve that approach to successfully operate on synthesized gate-level netlist [113]. Several logic encryption techniques have been introduced to protect the ICs against those reverse engineering techniques, such as *logic locking* and *gate-level camouflaging* [93] [123]. Gate-level camouflaging is a particular camouflaging technique that utilizes the camouflaged standard cells for technology mapping against imaging-based reverse engineering. Each camouflaged cell has several possible logic functions at the imaging level, but only one function is physically implemented [94][65]. Alternatively, logic encryption can be done by introducing new primary inputs as *key inputs*, as well as additional logic, to the original circuits. The correct output value can be computed only if the users are aware of the correct key values [8] [93] [123].

To evaluate the above mentioned encryption techniques, several reverse engineering techniques have been developed based on fault-analysis and formal methods, such as *Boolean Satisfiability* [95][112][41][69][131]. Specifically, SAT-based reverse engineering techniques have been demonstrated to be a significant threat to logic encryption. Subramanyan et. al [112] presented a decryption algorithm based on satisfiability checking that selects distinguishable input patterns to rule out the incorrect keys. Regarding the encrypted circuits using camouflaged cells, several oracle-guided SAT-based decamouflaging approaches have been developed, while the true functions of the encrypted circuits are assumed to be unknown [41][69]. The reason why SAT-based reverse engineering techniques are very efficient is that each SAT iteration rules out a significant portion (typically more than 99%) of *"wrong guesses"*. In this thesis, a reverse engineering approach of Galois Field arithmetic circuits based on computer algebraic method is proposed. This approach can extract the function of GF multipliers and the irreducible polynomials used for constructing the finite fields, when the binary encodings of the input and output bits are unknown. This is based on

analyzing the algebraic signatures (expressions) of the output bits of gate-level GF multipliers that is described in Chapter 7.

1.2.6 Overview of the thesis

This thesis is organized as follows:

- In Chapter 2, the formal methods that could be used to address the formal analysis problems of arithmetic circuits are reviewed. It includes the explanation of why the contemporary formal methods are limited by analyzing arithmetic circuits.

- Chapter 3 introduces the function extraction technique. It also includes results of evaluating contemporary formal methods and verification of large arithmetic circuits using function extraction.

- In Chapter 4, verification of sequential arithmetic circuits is addressed. Specifically, two redundant polynomials are introduced, vanishing polynomial and don't care polynomial, that significantly reduce the complexity of algebraic rewriting.

- To address the limitation of applying algebraic rewriting on heavily optimized arithmetic circuits, an algebraic rewriting technique based on And-Inv-Graph (AIG) is presented in Chapter 5. Specifically, this technique normalizes arbitrary gate-level circuits into a netlist with Boolean gates and HAs/FAs.

- The new canonical representation, *algebraic spectrum*, is introduced in Chapter 6. Using this representation, the word-level abstraction problem of arithmetic datapath is addressed.

- In Chapter 7, the parallel verification and reverse engineering of finite field arithmetic circuits have been addressed.

CHAPTER 2

BACKGROUND

This chapter reviews the contemporary formal methods including *canonical diagrams*, *Boolean satisfiability* (SAT), *satisfiability modulo theories* (SMT), and the method used in this thesis, *computer algebra approach*.

2.1 Canonical Diagrams

Several approaches have been proposed to check an arithmetic circuit against its functional specification. Different variants of canonical, graph-based representations have been proposed, including Binary Decision Diagrams (BDDs) [20], Binary Moment Diagrams (BMDs) [22] [26], Taylor Expansion Diagrams (TED) [29], and other hybrid diagrams. BDDs are the most extensively used canonical diagrams for synthesis and verification. An example of equivalence checking using BDD is shown in Figure 2.2. The Boolean function has been mapped into two different logic designs shown in Figure 2.2 (a) and (b). Since BDD is canonical with a respect fixed variable order, EC can be done by comparing the two BDDs of the two designs with a identical variable ordering. In this example, the variable is orders in $a \rightarrow b \rightarrow c$. If the BDDs are identical, we say the two designs are functional equivalent. Alternatively, equivalence checking can be done by checking if the BDD of a *miter* can be reduced to *zero-BDD*.

While BDDs have been used extensively in logic synthesis, their application to verification of arithmetic circuits is limited by the prohibitively high memory requirement for complex arithmetic circuits, such as multipliers. BDDs are being used, along

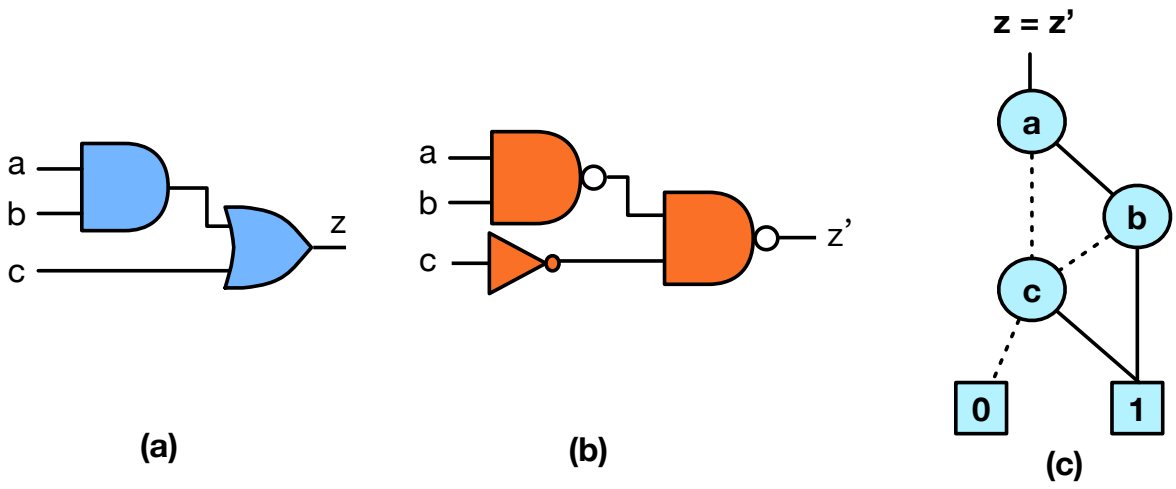


Figure 2.1: Canonical diagrams - Ordered Binary Decision Diagrams (OBDD); a) Gate-level design with output z; b) Gate-level design with output z'. c) The truth table of z and z'; d) BDDs of z and z' are identical with same variable order.

with many other methods, for local reasoning, but not as monolithic data structure [54]. BMDs and TEDs offer a better space complexity but require word-level information of the design, which is often not available or is hard to extract from bit-level netlists. While the canonical diagrams have been used extensively in logic synthesis, high-level synthesis, and verification, their application to verify large arithmetic circuits remains limited by the prohibitively high memory requirement of complex arithmetic circuits [31][128][70]. For example, multiplication is one of the examples that causes memory explosion problem. The BDD of a 4-bit integer multiplication is shown in Figure 2.2, generated using CUDD 2.4.0 package [107]. It includes 1,022 nodes. For a 6-bit multiplication, the number of BDD nodes is 8,176 and increases exponentially with the number of variables.

2.2 SAT and SMT solvers

Arithmetic verification problems can be modeled using Boolean satisfiability (SAT) or satisfiability modulo theories (SMT). Several SAT solvers have been developed to

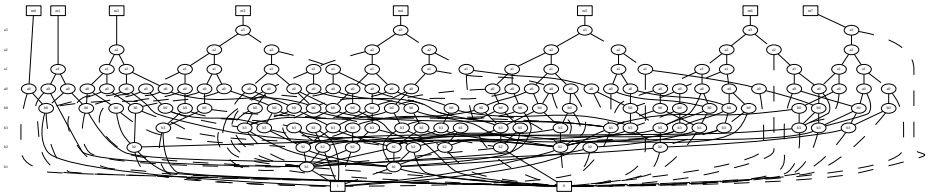


Figure 2.2: Reduced ordered binary decision diagrams of 4-bit multiplication.

solve Boolean decision problems, including ABC [75], MiniSAT [109], and others. Some of them, such as CryptoMiniSAT [108], target specifically XOR-rich circuits, but, like all others, are based on a computationally expensive Davis-Putnam-Logemann-Loveland (DPLL) decision procedure. Several techniques combine linear arithmetic constraints with Boolean SAT in a unified algebraic domain [42];

SMT solvers depart from treating the problem in a strictly Boolean domain and integrate different well-defined theories (Boolean logic, bit vectors, integer arithmetic, etc.) into a DPLL-style SAT decision procedure [16]. Some of the most effective SMT solvers, potentially applicable to our problem, are Boolector [82], Z3 [39], and CVC [6]. However, SMT solvers still model the problem as a decision problem and, as demonstrated by our experimental results, are not efficient at solving verification problems that appear in arithmetic circuits.

Boolean satisfiability checks if a given problem has a satisfiable solution. In Boolean logic, a formula is in conjunctive normal form (CNF) or clausal normal form which is a conjunction of clauses, where a clause is a disjunction of literals. The SAT solvers take a conjunction of clauses represented in a conjunction normal format (CNF) as input. It produces *UNSAT* if the problem is *unsatisfiable*, or produces a solution which satisfies the problem. Equivalence checking using SAT is mostly formulated using *miter* model (Figure 2.3). If the two designs are equivalent, *miter* will always evaluate to 0 (Boolean false). Hence, if two designs are equivalent, the result

returned by the SAT solver will be *UNSAT*. An example of a gate-level equivalence checking example using SAT solver is shown in Figure 2.3.

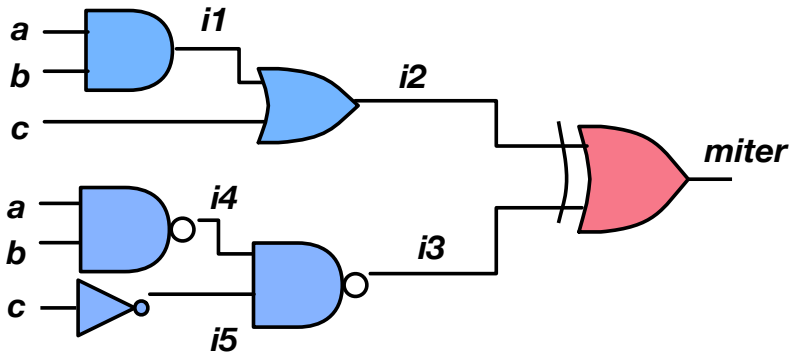


Figure 2.3: Example of equivalence checking using SAT solver using a Boolean *miter*.

The gate-level model of *miter* has to be translated into CNF first. The transformation between logic gates to CNF format is shown in Equation 2.1. After converting the model to a CNF, a unit clause (*miter*) will be added. The meaning of this unit clause is that given the condition that *miter* always evaluates to 1, SAT solvers checks whether there exist a satisfiable solution in this problem. If the result is *UNSAT*, it means they are equivalent, which means that *miter* always evaluates to 0. Otherwise, it returns a counterexample and returns *Satisfiable*. The same formulation can also be solved using SMT solvers. In addition to equivalence checking, this model has been applied to reverse engineering camouflaged circuits and solved by SAT solvers [69][132].

$$
\begin{aligned}
CNF(miter) &= (\bar{a} + \bar{b} + i_1)(a + \bar{i_1})(b + \bar{i_1}) \\
&\wedge (i_1 + c + \bar{i_2})(\bar{i_1} + i_2)(\bar{c} + i_2) \\
&\wedge (\bar{a} + \bar{b} + \bar{i_4})(a + i_4)(b + i_4) \\
&\wedge (\bar{c} + \bar{i_5})(c + i + 5) \\
&\wedge (\bar{i_4} + \bar{i_5} + \bar{i_3})(i_4 + i_3)(i_5 + i_4) \\
&\wedge (\bar{i_2} + \bar{i_3} + m\bar{i}ter)(i_2 + i_3 + m\bar{i}ter)(i_2 + \bar{i_3} + miter)(\bar{i_2} + i_3 + miter) \\
&\wedge (miter)
\end{aligned}
\tag{2.1}
$$

While SMT solvers support *bit-vector* operations, they allow formulating the logic in pseudo-Boolean expression. In other words, SMT solvers could solve a satisfiability problem of a word-level *miter*. For example, to check if a Half-adder (HA) shown in Figure 2.4 is correct, a word-level *miter* equals to *miter*=$a + b$-$(2c + s)$. The formulation for the logic gates is the same as the CNF as shown in the SAT example. Instead of adding a unit clause for the *miter*, a word-level *miter* with bit-vector formulation is required in this modeling. The formulation is shown in Equation 2.2. The last equation is the word-level *miter*. It models the miter as bit-vector adding $(a + b)$ and $(-2c - s)$. In the formulation, 10 is bit-vector constant 2. In Chapter III, I evaluate several *state-of-the-art* SAT solvers, and SMT solvers with two models, a Boolean model and a word-level model, using arithmetic benchmarks. It shows that these techniques cannot efficiently solve the verification problem of large arithmetic circuits.

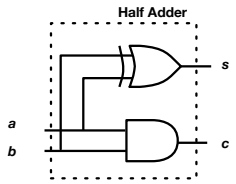


Figure 2.4: Half adder design with input a and b. s is the sum function and c is the carry function.

$$\begin{aligned}
&(\bar{a}+\bar{b}+\bar{s})(a+b+\bar{s})(a+\bar{b}+s)(\bar{a}+b+s) \\
&\wedge (a+b+\bar{c})(\bar{a}+c)(\bar{b}+c) \\
&\wedge (bvadd\ (bvadd(a,b)\ ,\ \neg bvadd(bvmul(10,c),s)))
\end{aligned} \tag{2.2}$$

2.3 Structural Minimization

Instead of solving the verification problem directly, there are many techniques developed to reduce the size of the problem first. Structural-based logic minimization technique based on logic rewriting on the graphs that describe the circuit structure has been demonstrated to be efficient in reducing the complexity of equivalence checking problem. Mishchenko et. al [76] presented an And-Inv-Graph (AIG) data structure that is the *state-of-the-art* technique for logic reduction and synthesis. Specifically, *Functionally-reduced AIG* (FRAIG) was developed for reducing the logic complexity that combines random simulation, SAT-based equivalence checking, and logic rewriting based on AIG. It is also known as equivalence nodes identification [78].

As an example, consider the circuit includes with AND gates, shown in Figure 2.5(a). First, logic minimization process identifies that the output function of gates B and C are identical (Figure 2.5(b)). Hence, the logic function can be minimized by eliminating the gate C and merging output z_1 with z_0. Note that this process aims to find a minimum representation of the Boolean function. The actual implementation

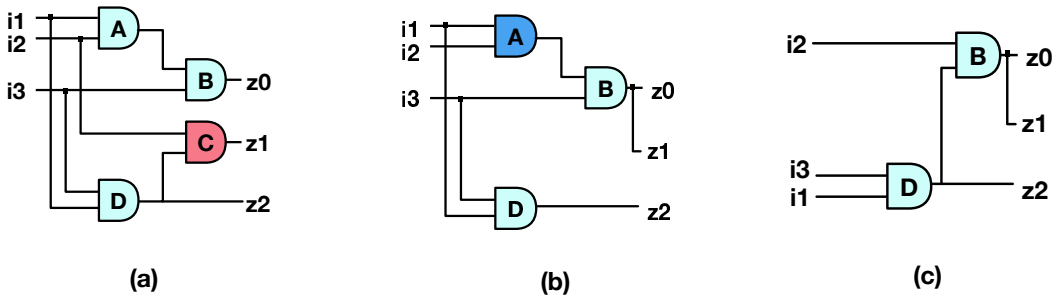


Figure 2.5: Example of logic minimization. a) Original circuit; b) Circuit with gates B and C are merged; c) Circuit with gates A and D are merged.

is not changed. Similarly, gate A can be eliminated since the output function is the same as gate D (Figure 2.5(c)). Hence, the original function can be represented using two AND functions instead of four AND functions. Assume to use SAT solver, this function is converted to CNF format with only six clauses instead of twelve clauses. However, this technique cannot efficiently reduce the complexity of combinational equivalence checking of non-linear arithmetic circuits. This is demonstrated in Chapter III Table 3.2 using a miter of two gate-level integer multipliers. It shows that the size of the *miter* is only reduced up to 0.1% using 64-bit or larger multipliers. Similarly technique for reducing the size of the SAT problems is also evaluated in Chapter III Table 3.2.

2.4 Computer Algebra Approaches

One of the most advanced techniques that have potential to solve the arithmetic verification problem are those based on symbolic Computer Algebra [38]. These methods model the arithmetic circuit specification and its hardware implementation as polynomials [96],[103],[122],[89],[71],[91]. The verification goal is to prove that implementation satisfies the specification by performing a series of divisions of the specification polynomial F by the implementation polynomials $B = \{f_1, \ldots, f_s\}$, representing components that implement the circuit. For example, the specification

of a multiplier circuit with word-level inputs X, Y and output Z is $F = Z - X \cdot Y$. The implementation polynomials are derived from gate equations, similar to those shown later in Equation(7.1).

To systematically perform polynomial division, term ordering ">" is imposed on monomials, so that each polynomial has a well defined leading term *lt()*. If polynomial f contains some term t that is divisible by the leading term $lt(g)$ of polynomial g, then the division of f by g gives a remainder polynomial $r = f - \frac{t}{lt(g)} \cdot g$. In this case, we say that f *reduces* to r modulo g, denoted $f \xrightarrow{g} r$. With this, the verification problem is posed as the reduction of F modulo B, denoted $F \xrightarrow{B}_{+} r$. The remainder r has the property that no term in r is divisible by the leading term of any polynomial f_i in B. The sign + refers to the fact that the division process is sequential, using polynomials of B one by one. Let $B = \{f_1, ...f_s\}$ be a set of polynomials representing circuit elements (logic gates, half adders, etc.) and let R be a polynomial ring, $R = \mathbb{F}\{x_1, ..., x_n\}$. The set of polynomials $B = \{f_1, \ldots, f_s\}$ generates an *ideal* $J = \langle f_1, ..., f_s \rangle$ with $f_i \in \mathbb{F}_q$, defined as: $J = \langle f_1, ..., f_s \rangle = h_1 f_1 + ... + h_s f_s \in \mathbb{F}_q$. The polynomials $f_1, ..., f_s$ are called the bases, or *generators*, of the ideal J. In the context of circuit verification, they model the *implementation* of the circuit.

In some cases, this test can be simplified to checking if $f \in I(V_{\mathbb{F}_q}) = J + J_0$, which is known in computer algebra as *ideal membership* testing. While an ideal J may have many different representations, the Grobner Basis of J is unique in order to a monomial order.

A standard procedure to test if $F \in J$ is to divide polynomial F by $f_1, ..., f_s$, one by one. The goal is to cancel, at each iteration, the leading term of F using one of the leading terms of $f_1, ..., f_s$. If the remainder of the division is $r = 0$, then F vanishes on $V(J)$, proving that the implementation satisfies the specification. However, if $r \neq 0$, such a conclusion cannot be made: B may not be sufficient to reduce F to 0, and yet the circuit may be correct. To check if F is reducible to zero one must use a *canonical*

set of generators, $G = \{g_1, ..., g_t\}$, called *Groebner basis*. Without Groebner basis one cannot answer the question whether $F \in J$.

Several algorithms have been developed to compute Groebner basis over the field, including the well known Buchberger's algorithm [23]. However, this algorithm is computationally expensive, as it computes the so-called *S-polynomials*, by performing expensive division operation on all pairs of polynomials in B. Even with newer algorithms, such as F4 [43], the computational complexity of Groebner basis computation remains prohibitively large for arithmetic circuits. Furthermore, what is the most important, these algorithms do not apply directly to rings over integers, $\mathbb{Z}_{2^n}$, which is needed to solve the verification problem for arithmetic circuits considered in this work. In general, this problem cannot be solved by testing if F is a member of an ideal $J = \langle f_1, ..., f_s \rangle$, i.e., if $F \in J$. Many of the results related to ideal membership that are valid over algebraically closed fields are fundamentally unsolved over integers $\mathbb{Z}$. It has been shown that solving the problem for $\mathbb{Z}_{2^n}$ requires testing if $F \in I(V(J))$, where $I(V(J))$ is a set of all polynomials that vanish on $V(J)$ [1] [71]. Unfortunately, except for some special cases (such as Galois fields, $\mathbb{F}_{2^k}$), it is not known what $I(V(J))$ is. Many of the results related to ideal membership that are valid over algebraically closed fields are fundamentally unsolved over integers $\mathbb{Z}$.

Wienand et. al. [122] model an arithmetic circuit as an *arithmetic bit-level* (ABL) network of adders and other arithmetic operators. Both the specification and the arithmetic operators are represented as polynomials over $\mathbb{Z}_{2^n}$. They show that, the properly ordered set G of polynomials representing logic gates automatically renders it a Groebner basis. The verification problem is solved by testing if specification F reduced modulo G vanishes over $\mathbb{Z}_{2^n}$ using a computer algebra system, SINGULAR [40]. In [89], the solution is further restricted to variables in $\mathbb{Z}_2$ and the reduction formulated directly over quotient ring $Q = \mathbb{Z}_{2^n}[X]/\langle x^2 - x \rangle$. Here, the ideal $\langle x^2 - x \rangle$ is the constraint restricting values of variables x to (0,1). While mathematically

elegant, adding this constraint for all variables makes the method computationally expensive for gate-level circuits. For this reason, the method of [89] is limited to ABL networks composed of half adders (HA). Unfortunately, it is not always possible to extract adders from a gate-level circuit, especially in highly bit-optimized implementations. For this reason, this method is not applicable to gate-level implementations, considered in this work.

Kalla, et. al [71][91][92][114], formulated the verification problem similarly, but applied it to Galois field (GF) arithmetic circuits, which enjoy certain simplifying properties. Specifically, for GF, the problem reduces to the ideal membership testing over a larger ideal that includes $J_0 = \langle x^2 - x \rangle$ in $\mathbb{F}_2$. The solution uses a modified Gaussian elimination technique. In [91], a symbolic computer algebra method is used to derive a word level abstraction for GF circuits, where GF operators are elements of a polynomial ring with coefficients in $\mathbb{F}_{2^k}$. This work relies on the customized computation of Groebner basis and applies only to GF networks. It does not extend to polynomial rings in integers $\mathbb{Z}_{2^n}$ which is the subject of this work.

A different approach to arithmetic verification has been proposed in work of Basith et. al. [7] and Ciesielski et. al. [30], where a bit-level network is described by a system of linear equations. The system is then reduced to a single *algebraic signature*, F_{Sig}, using standard linear algebra methods and compared to the specification polynomial F_{spec}. A non-zero *residual expression*, $RE = F_{Sig} - F_{spec}$, determines a potential mismatch between the implementation and the specification, indicating a potential design error. An additional step is needed to check if $RE = 0$, which may be as difficult as the original problem itself. Furthermore, this method can only handle networks with linear signatures. An attempt to use a different model [30], by viewing the computation performed by the circuit as a flow of binary data has not offered particular improvement; the issue of testing if $RE = 0$ was replaced by checking the relation between the fanouts and floating signals that correctly captured the

Boolean nature of circuit variables but still is applicable only to networks with linear input signatures. An extension to this work has been recently presented in [32], by computing input signature from the known output signature using a *network-flow approach*. This technique also relies on the half-adder (HA) based circuit structure and represents logic gates as elements of HAs. Logic gates that cannot be mapped into adders are represented a proper combination of HAs, with an unused output left as "floating". Additional constraint relating floating signals to fanouts in the circuit must be satisfied for the result to be trusted; however, the computation to verify this condition can be expensive. For this reason, this method becomes inefficient if the number of logic gates dominates the HA network. Also, the circuit would need to be partitioned into linear and non-linear portions, which is a non-trivial task.

In contrast, the technique described in this work targets on an arbitrary, unstructured gate-level arithmetic circuit without requiring any reference to higher level models such as adders; it can efficiently handle nonlinear circuits without a need to distinguish between linear and nonlinear parts.

In summary, the problem of formally verifying integer arithmetic circuits over integers $\mathbb{Z}_{2^n}$ remains open. Currently, there are no known mathematical solutions to this problem in $\mathbb{Z}_{2^n}$. The approaches discussed above that managed to reduce the verification problem to testing if $F \in J$ impose restrictions on the type of the circuits that they can handle [89] [71]. Others, such as [7] cannot properly model the inherently Boolean signals using algebraic models. To the best of our knowledge, the techniques reviewed here cannot efficiently solve the verification problem for gate-level arithmetic circuits in $\mathbb{Z}_{2^n}$ over Boolean variables $\mathbb{Z}_2$, which is the problem in this work.

The technique proposed here solves the functional verification problem by devising an alternative but equivalent method, based on polynomial substitution and elimination. It correctly implements ideal membership testing without a need for expensive

division process with Groebner basis. The results demonstrate that it scales better and is more efficient than the state-of-the-art computer algebra methods.

CHAPTER 3

FORMAL VERIFICATION OF INTEGER ARITHMETIC CIRCUITS USING FUNCTION EXTRACTION

3.1 Introduction

The chapter presents an algebraic approach to functional verification of gate-level, integer arithmetic circuits, called *function extraction*. The arithmetic verification is based on extracting a *unique* bit-level polynomial function implemented by the circuit, directly from its gate-level implementation. The method can be used to verify the arithmetic function computed by the circuit against its known specification, or to extract the arithmetic function implemented by the circuit. Experiments were performed on arithmetic circuits synthesized and mapped onto standard cells using ABC system. The results demonstrate scalability of the method to large arithmetic circuits, such as multipliers, multiply-accumulate, and other elements of arithmetic datapaths with up to 512-bit operands and over 2 million logic gates. The results show that our approach wins over the state-of-the-art SAT/SMT solvers by several orders of magnitude of CPU time. The procedure has linear runtime and memory complexity, measured by the number of logic gates.

3.2 Function Extraction

Function extraction is done by transforming the polynomial representing the encoding of the primary outputs (called the *output signature*) into a polynomial at the primary inputs (the *input signature*). If the specification of the circuit is known, the extracted input signature will be compared with that specification, and in case

of a mismatch, it will provide a counter-example (bug trace) [46]. Otherwise, the computed signature identifies the arithmetic function implemented by the circuit.

The method uses an algebraic model of the circuit, with logic gates represented by algebraic expressions, while correctly modeling signals as Boolean variables. In contrast to [32], it works directly on unstructured, gate-level implementations. And, in contrast to [89],[91] and other computer algebra methods, it is done using efficient polynomial transformation, without a need for expensive Groebner Basis based polynomial division.

To the best of our knowledge, this approach has not been attempted before in the context of gate-level integer arithmetic in $\mathbb{Z}_{2^n}$[1]. It provides a practical method for checking if the implementation satisfies the specification without resorting to the ideal membership testing in $\mathbb{Z}_{2^n}$.

3.2.1 Algebraic Model

The circuit is modeled as a network of logic elements of arbitrary complexity: basic logic gates (AND, OR, XOR, INV) and complex (AOI, OAI, etc.) standard cell gates obtained by synthesis and technology mapping. In fact, the proposed model admits a hybrid network, composed of an arbitrary collection of logic gates and bit-level arithmetic components. At one extreme, it can be a purely gate-level circuit; at the other, a network composed of arithmetic components only. Each logic element is modeled as a *pseudo-Boolean polynomial* f_i, with variables from $\mathbb{Z}_2$ (binary) and coefficients from $\mathbb{Z}_{2^n}$ (integers modulo 2^n). The following algebraic equations are used to describe basic logic gates:

---

[1]The functional abstraction technique described in [91] applies only to Galois field circuits and is based on polynomial reduction via Groebner basis.

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

In our model, the arithmetic function computed by the circuit is specified by two polynomials: an input signature and an output signature. The *input signature*, Sig_{in}, is a polynomial in primary input variables that uniquely represents the integer function computed by the circuit, i.e., its *specification*. For example, an n-bit binary adder with inputs $\{a_0, \cdots, a_{n-1}, b_0, \cdots, b_{n-1}\}$, is described by $Sig_{in} = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i$. Similarly, the input signature of a 2-bit signed multiplier, shown in Fig. 3.1, is $Sig_{in} = (-2a_1 + a_0)(-2b_1 + b_0) = 4a_1b_1 - 2a_0b_1 - 2a_1b_0 + a_0b_0$, etc. In our approach, the input specification need not to be known; it will be derived from the circuit implementation as part of the verification process.

Similarly, the *output signature*, Sig_{out}, of the circuit is defined as a polynomial in the primary output signals. Such a polynomial is uniquely determined by the n-bit encoding of the output, provided by the designer. For example, the output signature of the 2-bit signed multiplier in Fig. 3.1 is $-8z_3 + 4z_2 + 2z_1 + z_0$. In general, an output signature of an unsigned arithmetic circuit with n output bits z_i is represented as a linear polynomial, $Sig_{out} = \sum_{i=0}^{n-1} 2^i \ z_i$. Similar expression is derived for signed arithmetic circuits, with its most significant bit z_{n-1} having a negative coefficient -2^{n-1}.

Our goal is to transform the output signature, Sig_{out}, using polynomial representation of the internal logic elements, into the input signature, Sig_{in}. By construction, the resulting Sig_{in} will contain only the primary inputs (PI) and will uniquely determine the arithmetic function computed by the circuit (cf. Theorem 1 in Section 3.2.4).

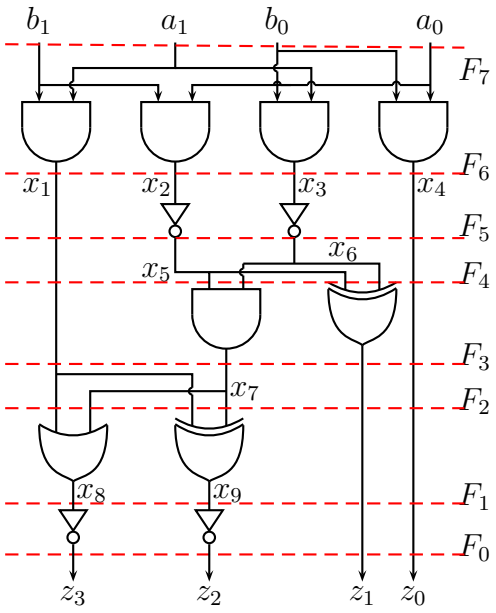


Figure 3.1: Verifying a 2-bit signed multiplier: Gate-level circuit with output signature $Sig_{out} = -8z_3 + 4z_2 + 2z_1 + z_0$.

3.2.2 Outline of the Approach

Algorithm 1 Verification Flow

Input: Gate-level netlist, output signature Sig_{out}
(input signature Sig_{in})
Output: $Pseudo - Boolean$ expression extracted by rewriting

1: Parse gate-level netlist; create algebraic equations for gates/modules
2: Find ordering for variable substitution (levelization, dependency)
3: $i = 0$; $F_i = Sig_{out}$
4: **while** there are unused equations **do**
5: Rewrite: $F_{i+1} = F_i$ with variables substituted with gate equations;
6: $i =$ i + 1
7: **end while**
8: **return** $F = F_i$ (to be compared with Sig_{in})

The proposed verification flow is outlined in Algorithm 3. The inputs to the algorithm are: the gate-level netlist (*implementation*); output signature Sig_{out} (*encoding* of the result at PO); and optionally the input signature Sig_{in} (*specification*). The first step is to translate the gate-level implementation into algebraic equations (*line* 1). Then, the algebraic equations are ordered according to the circuit structure and its

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

The rewriting process is illustrated with a simple 2-bit signed multiplier example, shown in Fig. 3.1. Each equation corresponds to a *cut* in the circuit, i.e., a set of signals that separate primary inputs from primary outputs; its pseudo-Boolean expression is denoted in the Figure by F_i.

First, F_0 is transformed into F_1 using substitutions $z_3 = 1 - x_8$ and $z_2 = 1 - x_9$. Subsequently, F_2 is obtained from F_1 using equations for x_8 and x_9, and so on, culminating at the primary inputs with expression $F_7 = 4a_1b_1 - 2a_0b_1 - 2a_1b_0 + a_0b_0$.

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

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

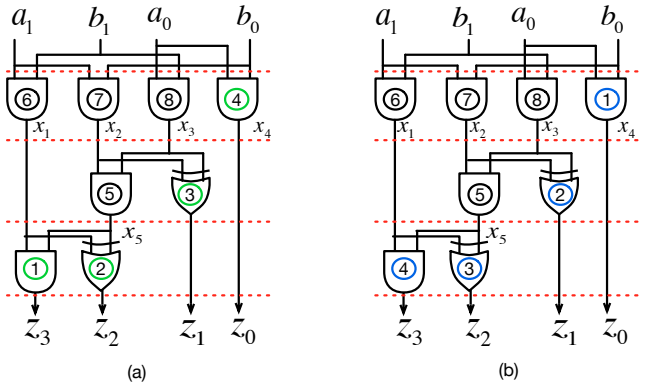


Figure 3.2: Two substitution orders for an unsigned 2-bit multiplier.

Table 3.1: 2-bit multiplier intermedia expression size of two substitution sequence

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

- **Substitution order:** The substitution order has the greatest influence on the intermediate expression size (the number of monomials). Even for a small difference between two orders, the maximum intermediate expression size may differ by several orders of magnitude larger in a large design. We illustrate the impact of the substitution order using a 2-bit multiplier (Figure 3.2). Orders (a) and (b) are two different substitution solutions which the first four iterations are different. We record the intermediate expression size step by step during rewriting (TABLE 3.1). We can see that order (b) experiences a larger peak

than order (a). We present two methods to find the efficient substitution order: *Dependency* and *Levelization*.

- **Dependency:** Substitution must follow the reversed-topological order; once a given variable (output of a gate) is substituted by an algebraic expression of the gate inputs, it will be eliminated from the current expression and will never be considered again. That is, a variable is substituted for only after substituting all signals in its logical cone. For example, in Figure 3.1, before substituting for x_6, one must substitute for x_7 and z_1, since they both depend on x_6. Otherwise, one will be forced to substitute again for the same variable(s) (in this case x_6) again later after substituting the signals in the cone below. Then, opportunity for early cancellations would be missed, leading to a potential computational explosion. Since the circuit is acyclic, there always exists an ordering of substitutions that satisfies this condition. We refer to this topological constraint informally as "vertical", since it orders variables upwards from POs to PIs.
- **Levelization:** To further increase the efficiency of substitution, a "horizontal" constraint is also imposed on the ordering of the candidate variables at a given transformation step. Specifically, the variables that are at the same logic level (from PIs) and have transitive fanin to common variables should be eliminated together, as this will maximize a chance of the reduction of common terms. It is these variables that define the best cut at each step of the procedure.

We demonstrate why substitution order greatly impacts the rewriting process using larger examples (Figure 3.3). We compare the rewriting process of 4-bit, 6-bit, and 8-bit CSA multipliers using *dependency* and *levelization*. In Figure 3.3, the x-axis represents the rewriting process in percentage of computation.

The y-axis represents the size of intermediate expression, i.e. the number of monomials in the expressions. We can see that the difference of the size of the intermediate expression using *dependency* and *levelization* increases when the size of the design is increasing. This means that the substitution order has greater impact on the rewriting process if the designs are more complex.

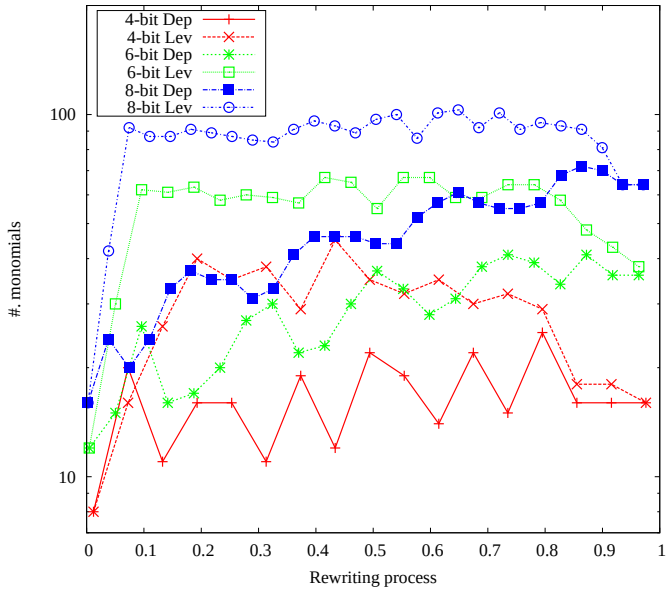


Figure 3.3: Substitution order analysis using 4-bit, 6-bit, and 8-bit multiplier. *Dep* is dependency; *Lev* is levelization.

- **Fanouts:** The size of the intermediate polynomial generated during rewriting can be reduced by identifying variables that depend on common inputs (fanouts of some variables). In this case, the substitution of such variables can be done simultaneously as this increases a chance for eliminating common subexpressions. For example, in Fig. 3.1 variables x_8, x_9 in subexpression $(8x_8 - 4x_9)$ of F_1 depend on common fanout variables x_1 and x_7. As a result, the subexpression $(8x_8 - 4x_9) = 4(2x_8 - x_9)$ reduces to $4(x_1 + x_7)$, without introducing a nonlinear term $8x_1x_7$, so F_1 can be directly transformed into F_3. Such nonlinear

terms are particularly harmful if their variables continue to be substituted by other variables, potentially leading to an exponential explosion.

Another simplification that can be applied during rewriting relies on recognizing some pre-defined multiple-input modules with known I/O signatures, such as half adder or full adder. Adders are particularly useful, since they exhibit linear relationship between their inputs and outputs. For example, the circuit in Fig. 3.1 contains a half adder with inputs x_5, x_6 and outputs x_7, z_1, with a linear I/O relationship described by $(x_5 + x_6 = 2x_7 + z_1)$. In this case, the subexpression $4x_7+2z_1$ in F_3 can be directly translated into $2(x_5+x_6)$, avoiding an intermediate nonlinear term $4x_5x_6$ of F_4. As a result, cut F_3 can be directly transformed into F_5.

In order to perform an efficient rewriting, we must analyze circuit topology to find the order will maximize the number of cancellations. The ordering algorithm must recognize reconvergent fanouts that can offer simplification of internal logic. For example, in the parallel prefix adder circuit, Fig. 3.4, both inputs of each OR gate are coming as reconvergent fanouts from a half adder. This effectively reduces the algebraic equation for OR from $a \vee b = a + b - ab$ to just $a + b$. For example, signals $x_5 = a_3 \wedge b_3$ and $x_8 = (a_3 \oplus b_3) \wedge x_3$ are coming from the *carry* (C) and *sum* (S) outputs of HA_3, so that $x_5x_8 = 0$. As a result, x_{11} at the output of OR_1 gate simplifies to $x_5 + x_8$.

- **Vanishing Polynomials:** In some arithmetic circuits a particular output bit may always evaluate to zero. This is typically associated with MSB, but this is not the only case. For example, in the squarer circuit ($Z = A^2$) the output bit z_1 is always 0. For this reason one may want to exclude bit z_1 from the output signature, $Sig_{out} = \sum_{i=0}^{n-1} 2^i \; z_i$. However, the set of algebraic expressions associated with the term $2z_1$ offers some early simplification during the computation

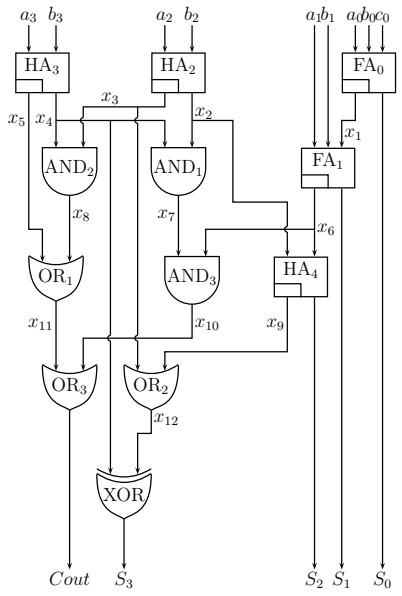


Figure 3.4: Parallel prefix adder, hybrid model

of the signature, before it reaches the primary inputs. Obviously, the logic cone of z_1 itself will reduce to 0 at the PI, but the terms of its intermediate cuts (at internal signals) help reduce the size of the intermediate cuts of the rest of the circuit. We refer to such a redundant expression as the *vanishing polynomial*, as it vanishes (evaluates to 0) for all possible values of its input variables. Note that the term *vanishing polynomial* has been used in [103] in a slightly different context.

- **Complex gates:** Our signature transformation algorithm works on a fabric of basic Boolean gates; this offers high logic granularity and the greatest choice of signals for the selection of the smallest cut. For the design with complex gates (standard cells AOI, OAI, etc.), algebraic equations are written for each internal signal of the gate, rather than only for its output. As confirmed by our experiments, this offers a richer set of cuts to choose from and increases a chance of an earlier simplification of the cut expression.

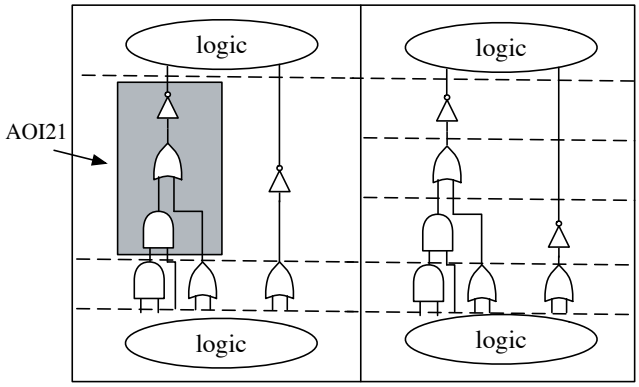


Figure 3.5: Expanding complex gates for cut rewriting.

- **Binary signals:** During elimination, the expensive division by the ideal $\langle x^2 - x \rangle$, employed by [89], is replaced by lowering x^k to x every time variable x is raised to higher degree during the substitution process. For example, if at any point an expression contains a term xyx or x^2y, it will be replaced by xy. With this, an expression, such as $xyx - yxy$, will immediately reduce to 0. This significantly simplifies the procedure, compared to the division by $\langle x^2 - x \rangle$. This approach has also been used in recent works [71, 91] that targeted GF circuits. The lack of this kind of simplification was the main reason for the existence of *residual expression* in earlier works that advocated algebraic approach [7].

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

It is essential to guarantee that the algebraic expressions of logic gates (eq. 7.1) correctly model Boolean signal variables. That is, the internal signal variables computed using those algebraic models must evaluate to exactly the same Boolean values as when using strictly Boolean methods, for all possible binary input combinations. With this, many potentially large algebraic subexpressions produced during the substitution will reduce to zero. This point can be illustrated with an example of the OR_1 gate with output x_{11} in the 3-bit adder in Fig. 3.4, now written in algebraic rather than Boolean form (Figure 3.6). As one can see, the value of x_{11} is exactly the same as the one obtained above using strictly Boolean methods (where x_5x_8 was also shown to reduce to 0).

$$\textbf{\textit{Algebraic}} \begin{cases} x_{11} = x_5 + x_8 - (x_5x_8) \\ x_5 = a_3b_3 \\ x_8 = a_3 + b_3 - 2a_3b_3 \\ x_5x_8 = (a_3^2b_3 + a_3b_3^2 - 2a_3^2b_3^2)x_3 \\ \qquad = (a_3b_3 + a_3b_3 - 2a_3b_3)x_3 = 0 \\ x_{11} = x_5 + x_8 \end{cases}$$

$$\textbf{\textit{Boolean}} \begin{cases} x_5x_8 \Rightarrow x_5 \wedge x_8 \\ = (a_3 \wedge b_3) \wedge [\,(\neg a_3 \wedge b_3) \vee (a_3 \wedge \neg b_3)\,] \\ = [\,(a_3 \wedge b_3) \wedge (\neg a_3 \wedge b_3)\,] \vee [\,(a_3 \wedge b_3) \wedge (a_3 \wedge \neg b_3)] \\ = [(a_3 \wedge \neg a_3 \wedge b_3)] \vee [(a_3 \wedge b_3 \wedge \neg b_3)] \\ = 0 \end{cases}$$

Figure 3.6: Proof that x_5x_8 evaluates to 0 using both, the computer algebraic and Boolean methods.

3.2.3 Function extraction vs. Polynomial Division

As mentioned in Chapter II, the standard polynomial division based method is limited to the high complex mathematic computation if the remainder of the division $r \neq 0$, i.e., B may not be sufficient to reduce F to 0, and yet the circuit may be correct. This is also called a non-zero *residual expression*. To check if this type of

polynomial is reducible to zero, *Groebner basis* has to be applied. One of the main advantages of function extraction technique is that, residual expression never appears during the rewriting process.

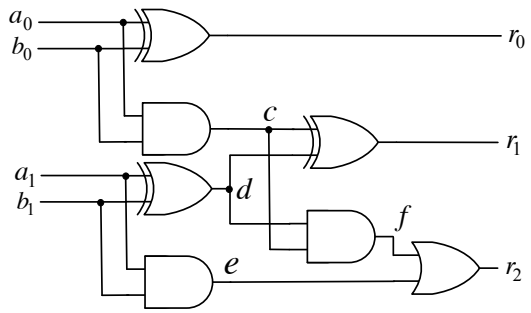


Figure 3.7: 2-bit gate-level adder. $R = r_0 + 2r_1 + 4r_2$, $A = a_0 + 2a_1$, $B = b_0 + b_1$. $R = A + B$.

$$
\begin{aligned}
&g_1 : r_0 - (a_0 + b_0 - 2a_0b_0) \\
&g_2 : c - (a_0b_0) \\
&g_3 : d - (a_1 + b_1 - 2a_1b_1) \\
&g_4 : r_1 - (c + d - 2cd) \\
&g_5 : f - (cd) \\
&g_6 : e - (a_1b_1) \\
&g_7 : r_2 - (e + f - ef)
\end{aligned}
\tag{3.2}
$$

To demonstrate the non-zero remainder issue of polynomial division method, a complete polynomial division process of a 2-bit gate-level adder is provided. The gate-level implementation is shown in Figure 3.7. The specification is defined as $F_{spec} = (A + B) - R$. The set of polynomials, B, which describe the implementation is shown in Equation 3.2.

$$
\begin{aligned}
F_0 &= (a_0 + b_0 + 2a_1 + 2b_1) - (4r_2 + 2r_1 + r_0) \\
F_0 &\xrightarrow{g_1} = F_1 = 2a_0b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 \\
F_1 &\xrightarrow{g_2} = F_2 = 2c + 2a_1 + 2b_1 - 4r_2 - 2r_1 \\
F_2 &\xrightarrow{g_3} = F_3 = 4a_1b_1 + 2d + 2c - 4r_2 - 2r_1 \\
F_3 &\xrightarrow{g_4} = F_4 = 4cd + 4a_1b_1 - 4r_2 \\
F_4 &\xrightarrow{g_5} = F_5 = 4f + 4a_1b_1 - 4r_2 \\
F_5 &\xrightarrow{g_6} = F_6 = 4e + 4f - 4r_2 \\
F_6 &\xrightarrow{g_7} = F_7 = 4ef
\end{aligned}
\tag{3.3}
$$

If the circuit is correct, then F should be reduced to 0 after dividing all the polynomials in B. The complete polynomial division process is shown in Equation 3.3. First, F_0=F_{spec} is divided using polynomial g_1 (see Equation 3.2), resulting in the intermediate specification $F_1 = 2c + 2a_1 + 2b_1 - 4r_2 - 2r_1$. The intermediate specification will be divided using all the polynomials in B. At each division step, we can see that there are new monomials introduced, and some monomials are eliminated. However, after dividing g_7, monomial $4ef$ is returned as the remainder.

To check whether the remainder $4ef$ can be reduced to zero, polynomial division has to be applied on the remainder. This process is shown in Equation 3.4. At the fifth step of this division process, this remainder can be proved to be zero since all the variables in the polynomials are Boolean signals. Hence, $(a_1b_1a_1 + a_1b_1b_1 - 2a_1b_1a_1b_1)$ can be reduced to $(a_1b_1 + a_1b_1 - 2a_1b_1)$, where a_1^2 equals to a_1 in Boolean domain. However, this process requires complex mathematic computations for computing Gronerber Basis. For example, several *vanishing polynomials* have to be added in B to make sure that all the signals in the circuit are in Boolean domain, such as $a_1^2 - a_1 = 0$ and $a_0^2 - a_0 = 0$. Alternatively, $4ef = 0$ can be proved by rewriting the algebraic

or Boolean equations by subsituting the variables. For example, by rewriting the Boolean equations, we can prove $e \wedge f$ is always $false$, which indicates that ef is $false$. Hence, $4ef$ is 0.

$$\begin{aligned} R &= 4ef \\ &= 4e(cd) \\ &= 4(a_1b_1)(cd) \\ &= 4(a_1b_1)(a_1 + b_1 - 2a_1b_1)(a_0b_0) \\ &= 4(\underline{a_1b_1a_1 + a_1b_1b_1 - 2a_1b_1a_1b_1})(a_0b_0) \\ &= 4(\underline{0})(a_0b_0) \\ &= 0 \end{aligned} \tag{3.4}$$

3.2.4 Properties of Computed Input Signature

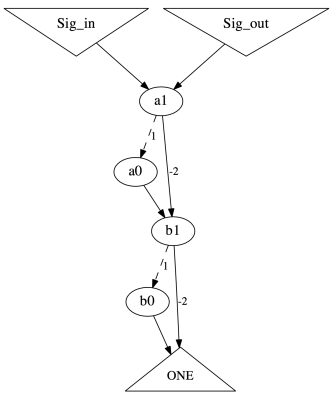


Figure 3.8: Arithmetic function of a 2-bit multiplier extracted from the circuit using TED in normal factored form: $Sig_{in} = (-2a_1 + a_0)(-2b_1 + b_0)$.

Once Sig_{in} has been computed, it is analyzed to see if it matches the expected specification. The comparison between the two expressions can be done using canonical data structures, such as BMD [22] or TED [29] that can check equivalence between

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

If the specification is not given, TED can provide the function implemented by the circuit in normal factored form to help identify the type of arithmetic function obtained. TED has a capability of finding the ordering of variables from which such a form can be obtained [28]. In large arithmetic circuits, additional variable ordering directives may be given by the designer if the bit-level composition of input words is known. For example, for the circuit in Fig. 3.1, the input signature computed by our method is $Sig_{in} = 4a_1b_1 - 2a_0b_1 - 2a_1b_0 + a_0b_0$. Its TED representation shown in Fig. 3.8 reveals the canonical factored form, $Sig_{in} = (-2a_1 + a_0)(-2b_1 + b_0)$. This indicates that the function computed by the circuit is a two-bit signed multiplier, $A \cdot B$, where the variables (a_1, a_0) and (b_1, b_0) form the two-bit input words, A and B.

Essential part of the described approach is the following theoretical result about the correctness and uniqueness of the computed input signature.

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

Proof: The proof of correctness hinges on the fact that each internal signal is correctly represented by an algebraic expression, i.e., such an expression evaluates to a *correct Boolean value*. Specifically, it can be easily verified that equations (7.1) are the correct algebraic representations of basic Boolean functions. Hence, any logic function that is expressed recursively by Eq. (7.1) must evaluate to a correct Boolean value;

and once the polynomial is reduced by removing redundant terms, the algebraic representation is unique. Example: XOR function, $f = a \oplus b = a'b + ab'$, can be written as $f = (1-a)b + a(1-b) - ((1-a)b)(a(1-b))$, which reduces to a unique form, $a+b-2ab$. Hence, a PO signal is correctly represented by variables in its logic cone, up to the primary inputs. Therefore, Sig_{out}, which is the weighted sum of the output signals, is eventually replaced by Sig_{in}. For this reason such computed Sig_{in} is a correct algebraic representation of the circuit.

The proof of uniqueness is done by induction on i, the step when polynomial F_i is transformed into F_{i+1}. Base case: polynomial $F_0 = Sig_{out}$ is unique. Also, as discussed above, algebraic representation of each logic gate is unique.

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

This theorem applies to combinational circuits, but it can be readily extended to *sequential circuits* by unrolling the circuit over a fixed number of time frames into a combinational circuit (bounded model).

3.3 Experimental Results

3.3.1 Comparison with SAT and SMT

The function extraction technique described in this chapter was implemented in C++. It performs rudimentary variable substitution and elimination, using the ordering strategy and implementation discussed in Section 3.2.2. The program was tested on a number of gate-level combinational arithmetic circuits, taken from [61]: CSA

multipliers, add-multiply, matrix multipliers, squaring, etc., with operands ranging from 64 to 512 bits. The results are shown in Tables 3.3 and 3.4. The experiments were conducted on a PC with Intel Processor Core i5-3470 CPU 3.20GHz x4 with 15.6 GB memory. The gate-level structures were obtained by direct translation of standard implementation of the designs onto basic logic gates [61]. The designs labeled with extension *.syn* were synthesized and mapped using ABC system [75] (commands: *strash; logic; map*) onto *mcnc.genlib* standard cell library. The plot for CPU runtime in Fig. 3.10 a) shows an approximately *linear* runtime complexity of the program in the number of gates for all the tested circuits. This should be contrasted with quadratic runtime complexity of [32] (col. 5) and the exponential time complexity of other tools.

As proposed in Section 3.2.2, the reason why our technique is efficient is that rewriting the Sig_{out} provides significant internal expression elimination. We demonstrate this by measuring the size of the internal expressions of Sig_{out} and the individual output bit expressions (Figure 3.11). We can see that the expressions for z_5, z_6, z_7 are more than 100 times larger than the Sig_{out} in the middle of the rewriting process. However, each output bit expression contains many common monomials which can be eliminated by weighted addition (i.e. Sig_{out}).

3.3.1.1 SAT comparison:

We tested the applicability of SAT tools to the the type of arithmetic verification problems described in this chapter. The functional verification problem was modeled as a combinational equivalence checking problem, generated with a miter using ABC (command *miter*)[75], with the reference design generated by ABC using [*gen -N -m*] command. Then, we check if the miter is unsatisfiable. The state-of-the-art SAT solvers were tested using the CNF files created by ABC. ABC was also tested using the combinational equivalence checking *cec* (Table 3.4).

The *CEC* approach in ABC is based on AIG rewriting via structural hashing, simulation and the state-of-the-art SAT [79]. This technique reduces the overall complexity of checking equivalence between two designs by finding equivalent internal AIG nodes. However, finding internal equivalent nodes in non-linear arithmetic designs is very difficult. In Table 3.2, N_1, N_2 are the numbers of AIG nodes before and after function reduction [$fraig - v$] [75]). Δ_1 shows the percentage of reduced nodes. The reference design is generated by ABC [*gen -N -m*] command. We can see that *fraig* is unable to identify and merge the internal equivalent nodes. Additionally, we evaluate the complexity of checking *Satisfiability* using SAT solver *lingeling* [12]. N_3, N_4 are the numbers of clauses before and after simplification by [12]. Δ_2 shows the percentage of reduced clauses. We can see that both *fraig* and SAT solver cannot simplify the integer multiplier CEC problem. For this reason, such techniques are inefficient to verify non-linear arithmetic gate-level designs.

We also tested the SAT-based pseudo-Boolean solvers *MiniSat+* [110] and *PB-Sugar* [81] that have been applied to problems dealing with large pseudo-Boolean expressions. The specification is modeled as a pseudo-Boolean expression ($Sig_{out} - Sig_{in}$) and the gate-level implementation using the algebraic model, as in Eq. (1). If such constructed problem is *unSAT*, the implementation is bug-free. Both solvers successfully verified a 4-bit CSA multiplier, but were unable to solve the problem for a CSA multiplier circuit greater than six bits in 24 hours.

Table 3.2: N_1, N_2 are the numbers of nodes before and after $fraig$ *-v* in *ABC*; N_3, N_4 are the numbers of clauses before and after simplification by [12]

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

3.3.1.2 SMT experiments

Given the specification Sig_{in} and output encoding Sig_{out}, the goal was to prove that $(Sig_{out} - Sig_{in})$ is unsatisfiable (unSAT). Two types of modeling of the gate equations were tested:

- **SMT Model 1:** We directly translated the algebraic equations of the gate-level implementations into SMT2 format and modeled the specification (Sig_{out}-Sig_{in}) as a Pseudo-Boolean polynomial using Boolean vector operations.
- **SMT Model 2:** The product circuit (miter) was translated directly into SAT by converting the CNF model into SMT2 format. The CNF files used in this experiment were the same as input to the SAT experiments. The second approach showed better performance; it is the one shown in Table 3.4.

Table 3.4 gives comparison of our results for the synthesized multipliers with winners of 2015 SMT competitions and evaluation, including Boolector [82], Z3 [39], CVC4 [6]; minisat_blbd [11], lingeling [12] and the ABC system [75]; with the symbolic algebra tool, SINGULAR [40]; and Synopsys' *Formality* system. It shows that our technique surpasses those tools in CPU time by several orders of magnitude.

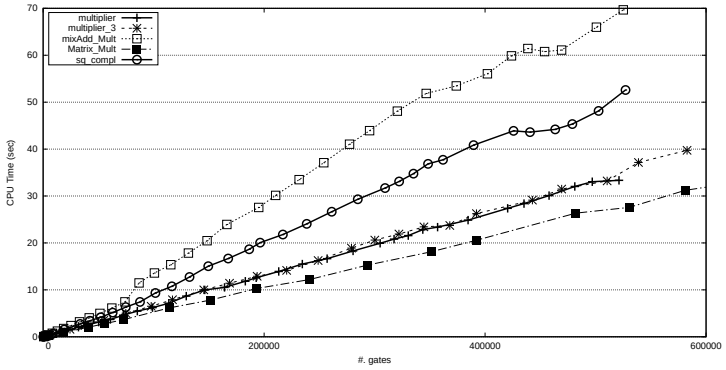


Figure 3.9: Verifying combinational arithmetic circuits: CPU time.

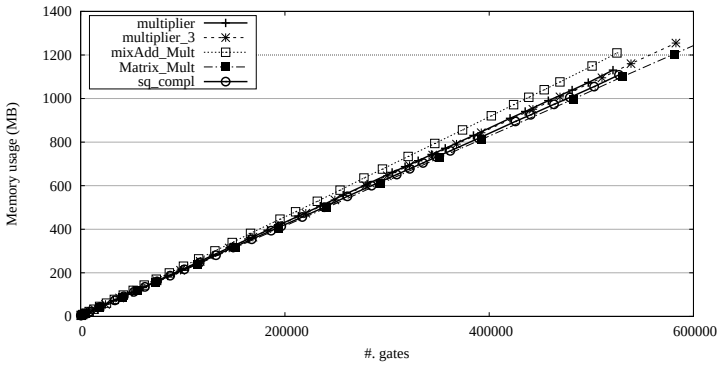


Figure 3.10: Verifying combinational arithmetic circuits: Memory usage.

Table 3.3: CPU time and memory results of 256-bit (Operands A and B) arithmetic circuits. (TO = timeout after 3600 sec; MO = memory out of 8 GB).

Benchmark		256-bit		
Name	Function	# Gates	CPU [sec]	MEM
adder	$F = A + B$	1.8K	0.10	5.7 MB
adder_syn	$F = A + B$	1.8K	0.19	6.4 MB
shift_add	$F = A + A/2$	7.7K	0.44	18.2 MB
multiplier	$F = A \times B$	521K	32.26	1.15 GB
multiplier_syn	$F = A \times B$	663K	285.22	1.25 GB
mixAddMult	$F = A \times (B+C)$	525K	70.18	1.18 GB
mixAddMult_syn	$F = A \times (B+C)$	650K	209.31	1.12 GN
multiplier_3	$F = A_1 \times B_1 + A_2 \times B_2 + A_3 \times B_3$	1,571K	-	MO
sq_comp	$F = A^2 + 2A + 1$	527K	48.84	1.13 GB
cube_comp	$F = 1 + A + A^2 + A^3$	1,576K	TO	-
Matrix_Mult	$F = A[3\times 3] \times B[3\times 1]$	4,712K	-	MO

Table 3.4: Results for a synthesized multiplier; comparison with [32], SAT, SMT, and commercial tools (TO = timeout after 3600 sec; UD = undecided; MO = memory out of 8 GB). *ABC was unable to synthesize the 512-bit CSA multiplier due to memory limit.

multiplier_synthesized											
Statistics		This work		[32]	SAT [sec]			SMT [sec]			Commercial
Size	#Gates	CPU	Mem	[sec]	[12]	[25]	ABC	[82]	[39]	CVC4	[116]
4	86	0.01	2.2MB	0.45	0.00	0.00	0.01	0.00	0.03	0.09	0.81
8	481	0.04	2.9MB	1.72	4.40	62.75	11.66	7.18	16.55	42.63	3.19
12	1.2K	0.08	4.3MB	5.21	TO	1615.47	UD	2030.19	TO	TO	108.1
16	2.1K	0.14	6.1MB	7.34	TO	TO	UD	TO	TO	TO	111.2
64	41.4K	5.50	76MB	TO	TO	TO	UD	TO	TO	TO	675.4
128	164K	39.64	299MB	TO	TO	TO	UD	TO	TO	TO	TO
256	663K	285.22	1.3GB	TO	TO	TO	UD	TO	TO	TO	TO
512*	2,091K	130.22	4.4GB	TO	TO	TO	UD	TO	TO	TO	TO

3.3.2 Limitations and Proposed Solutions

3.3.2.1 Circuit Boundaries

Currently, the described method of functional verification by signature rewriting requires knowledge of the I/O boundary of the circuit. Specifically, we need to know

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

3.3.2.2 Output Encoding

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

The results shown in Figure 3.11 represent the intermediate expression size of individual output bit of a 4-bit multiplier. The horizontal axis represents the iteration number of rewriting process; the vertical axis represents the size of the expression at each point of the computation. We can see that the complexity of rewriting individual bits is different. The intermediate expression size of the 2^{nd} MSB is characterized by the highest complexity and LSB is the lowest one. The complexity of the individual output bits increases from z_0 to z_5. Based on this observation, we can determine the output encoding by monitoring the intermediate expression. The output bits close to MSB are very difficult to be extracted individually by our technique. The reason is that the intermediate expression is too large since there are few cancellations without the expressions from other outputs. However, to determine the output encoding, it is

not necessary to rewrite the signature all the way to the primary inputs. The output encoding can be determined earlier in the process and the process will be terminated immediately. For example, in Figure 3.11, all the output bits can be recognized before iteration #35.

3.3.2.3 Effects of Synthesis on Function Extraction

The performance of our technique is sensitive to logic synthesis and technology mapping. In Figure 3.12, we compare the rewriting process with different logic synthesis techniques using 8-bit CSA Multiplier. The horizontal axis represents the rewriting process as percentage of the complete run; the vertical axis represents the size of the expression at each point of the computation. *Original* presents the rewriting process of 8-bit multiplier without any optimization. In Figure 3.12, curves *resyn* and *resyn3* are two different logic synthesis commands provided by ABC; curve *complex* refers to the mapping library includes the complex gates (e.g. AOI21, OAI221, etc.); curve "no-complex" refers to the library contains only 2 input logic gates.

We can see that the original 8-bit multiplier provides a much lower intermediate expression size, which means that it is the easiest one to be verified. Synthesized multipliers mapped into complex gates are more difficult to verify than those with the simple gates. The reason why the intermediate expression size is larger is that the logic synthesis technique and technology mapping techniques *re-construct* the circuits. This creates fewer possible cancellations in our rewriting technique. Using the heuristics proposed in Section 3.2.2, we can verify a lightly-synthesized multiplier up to 256 (TABLE 3.4). However, the bit-optimized arithmetic design produced by *DesignCompiler* or ABC *dch* remains challenge for this method.

3.4 Verification of Datapaths - A Case Study

This technique has been applied on datapath verification. Most of the work in RTL verification concentrates on verifying translation from high level specification

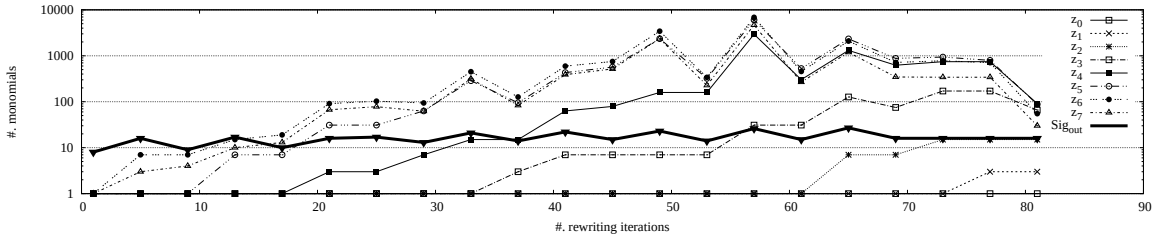


Figure 3.11: Comparing rewriting of the expression Sig_{out} vs individual output bits for a 4-bit multiplier.

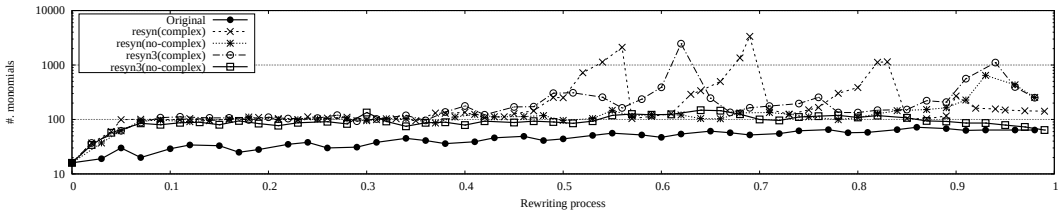


Figure 3.12: Synthesis impacts on function extraction

(such as C) to RTL [59]; some use DFG as a formal model for high-level specification [60]. Others use RTL to TLM abstraction for redesign and verification of RTL IPs [17]. Assertion-based verification technique (ABV) is also used for system-level design [106]. Industrial work in RTL verification typically addresses verification of RTL protocol implementation against its specification and uses TLC or TLA specification languages [10]. In this section, a case study of datapath verification using both word-level and bit-level verification method. Note that we consider all the datapaths are designed without truncation [111].

Consider an integer arithmetic logic unit (ALU), shown in Figure 3.13, taken from [117]. This architecture is used often in implementing integer operations for standard graphics APIs. The design consists of three word-level n-bit inputs, $A,\ B,\ C$, representing unsigned integers. Each of the operands can be optionally negated under the control of single-bit signals, neg_A, neg_B, neg_C. These bits, together with other configuration bits (en_{ab}, en_c and a negation bit neg_y of one of the local outputs), provide

control for various arithmetic functions, namely: $A \cdot B,\ -A \cdot B,\ A \cdot B + C,\ A \cdot B - C$, etc.

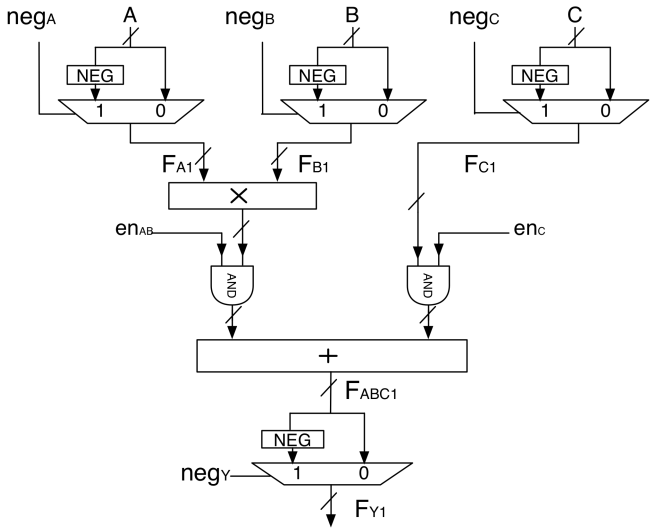


Figure 3.13: Integer ALU - initial RTL design

In [117] the integer ALU design was subjected to a number of algebraic and Boolean transformations resulting in the modified design shown in Figure 3.14. While the applied transformations can be shown to be mathematically correct, it is important to formally verify if the resulting RTL hardware implementation is indeed equivalent to the original one. This must be done to ensure that some unexpected bugs, typically related to finite bit-widths, sign extension, or two's complement implementation of subtraction, did not creep up into the final implementation. In [117] this problem was solved for $n = 16$ using *Hector*, a formal equivalence checking tool from Synopsys. The approach taken there required case-splitting and separately solving a number of individual cases, determined by the combination of the control signals.

We approach this problem differently and perform verification using symbolic representation for both RTL designs to verify if they are equivalent. Two versions of the proof are considered here: 1) In the first method the symbolic equations are derived for each design and a canonical TED representation [29] is used to show

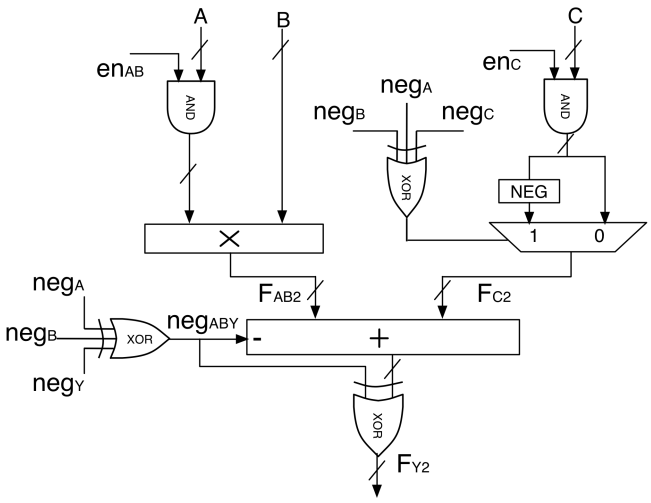


Figure 3.14: Integer ALU - final RTL design

that the two RTL implementations are equivalent for *arbitrary* bit-width, while still considering two's complement representation for negative numbers; and 2) A more convincing method considers a bit-level composition of the RTL structure and shows that the equivalence can be proven for large operand bit-widths [124]. Note: The *bit-level* RTL structure should not be confused with a *gate-level model*, since the arithmetic and logic operators are still defined on the register transfer level. In a separate model we can also demonstrate the correctness of our approach to gate-level designs.

3.4.1 Word-level Verification

In this model, the unsigned word-level operand X is represented simply as variable X (positive number) or as $-X$ (negative number), regardless of the number of bits (assuming no overflow).

- **Original Design** (F_{Y1}): The first level includes three identical modules, each composed of a *negator* (NEG) and a multiplexer (MUX) to select the operand in

a positive or in a negated form. The output of the first MUX, associated with operand A, is

$$F_{A1} = (1 - neg_A)A + neg_A(-A) = A \cdot (1 - 2 \cdot neg_A)$$

Note that for $neg_A = 0$, $F_{A1} = A$; and for $neg_A = 1$, $F_{A1} = -A$, as required. Similar expressions are derived for modules with inputs B and C, and outputs F_{B1} and F_{C1}. Next design level includes a multiplier followed by an enable signal en_{AB}, producing $F_{ABen1} = en_{AB}(F_{A1} \cdot F_{B1})$ and $F_{Cen1} = en_C \cdot F_{C1}$.

The next level has an adder with inputs $F_{AB1},\ F_{C1}$

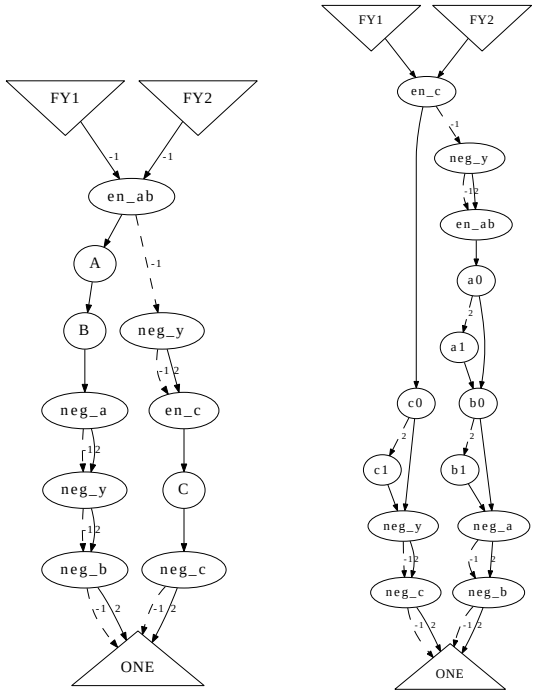
$$F_{ABC1} = F_{AB1} + F_{C1}$$

The lowest level has a negator gate for F_{ABC1}, controlled by neg_Y

$$F_{Y1} = F_{ABC1}(1 - 2 \cdot neg_Y)$$

The entire set of such equations is written into the TDS system [28] and represented by a canonical, word-level diagram, TED [29]. The diagram automatically represents the function in terms of the primary input variables, as shown in Figure 3.15(a). Note that Figure 3.15(a) also contains the diagram of the final design F_{Y2}, which is functionally the same as F_{Y1}. We can see that the equivalence has been proved by TED by constructing two functions in the same diagram. However, to prove the equivalence using rewriting method, we need to extract the polynomial expression of F_{Y2} from PO to PI.

- **Final Design (F_{Y2})**: The transformed design is shown in Figure 3.14, where



(a) (b)

Figure 3.15: TED representation: (a) word-level model; (b) bit-level model

$$neg_{ABC} = neg_A \oplus neg_B \oplus neg_C$$

$$neg_{ABY} = neg_A \oplus neg_B \oplus neg_Y$$

Translation of the Boolean operator $\oplus$ (XOR) into an algebraic expression can be done using the following, well known relation:

$$x \oplus y = x + y - 2 \cdot x \cdot y \tag{3.5}$$

By applying this formula to the above equations, we obtain:

$$neg_{AB} = neg_A + neg_B - 2 \cdot neg_A \cdot neg_B$$

$$neg_{ABC} = neg_{AB} + neg_C - 2 \cdot neg_{AB} \cdot neg_C$$

$$neg_{ABY} = neg_{AB} + neg_Y - 2 \cdot neg_{AB} \cdot neg_Y$$

With this, the remaining part of the design can described by the following set of expressions:

$$F_{Aen2} = en_{AB} \cdot A$$

$$F_{AB2} = F_{Aen2} \cdot B$$

$$F_{Cen2} = en_C \cdot C$$

$$F_{C2} = C \cdot (1 - 2 \cdot neg_{ABC})$$

$$F_{ABC2} = F_{AB2} + F_{C2} - neg_{ABY}$$

where neg_{ABY} is an integer *binary* variable. The same signal is then applied as a Boolean signal to an XOR to conditionally flip the bits of the word-level signal F_{ABC2} computed by the adder. The algebraic model for XOR shown in (3.5) does not apply to such bit-wise operation on a word-level signal, and needs to be suitably modified. Specifically, it can be modeled as a MUX, shown in Figure 3.16. When $neg_{ABY} = 0$, the output of the adder, ($F_{ABC2} = F_{AB2} + F_{C2} - 0$), is passed directly to the output F_{Y1}; and when $neg_{ABY} = 1$, the adder's output, ($F_{AB2} + F_{C2} - 1$), is bit-wise complemented. To model this, we use the standard relation between the bit-wise complement and a word-level complement/negation, $-X = \overline{X} + 1$. This, as already shown in [117], can be rewritten as $-(X - 1) = \overline{X - 1} + 1$, which, in turn, implies that

$$-X = \overline{X - 1}$$

With this we can now model the XOR and a MUX with inputs X and $-X$, where $X = F_{AB2} + F_{C2}$, as follows:

$$F_{Y2} = (1 - neg_{ABY})X + neg_{ABY}(-X) = X(1 - 2 \cdot neg_{ABY})$$

which is similar to the negator developed earlier.

Substituting $X = F_{AB2} + F_{C2}$ in the above equation gives the following model for the resulting MUX (c.f. Figure 3.16).

$$F_{Y2} = (F_{AB2} + F_{C2})(1 - 2 \cdot neg_{ABY})$$

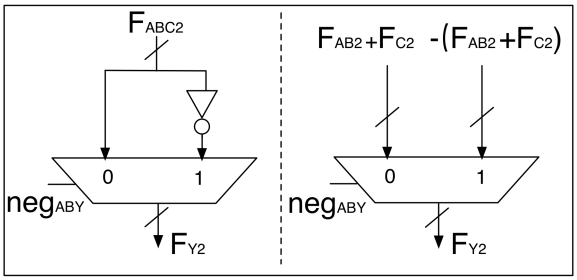


Figure 3.16: Modeling of the word-level XOR.

As mentioned in Item 1, the proof of equivalence between F_{Y1} and F_{Y2} using TED is shown in Figure 3.15. We can also prove that they are equivalent by comparing the extracted polynomial expressions. $F_{Y1} = F_{AB}(1-2 \cdot neg_Y)+F_C(1-2 \cdot neg_Y)$, and F_{Y2}=$(F_{AB} + F_C)(1 - 2 \cdot neg_Y)$. It is obvious that the expressions of F_{Y1} and F_{Y2} are the same.

3.4.2 Bit-level Verification

To perform RTL verification on a bit-level, we must consider the bit composition of each of the word-level signals. This is done by expressing each n-bit unsigned number X by its binary encoding: $X = \sum_{i=0}^{n-1} 2^i x_i$. The negative number $(-X)$ is represented using two's complement model as $-X = 2^n - X$. Specifically, we express the word-level input A using binary encoding

$$A = 2^{n-1}a_{n-1} + \cdots + 2a_1 + a0$$

and similarly for inputs B and C. The system of equations derived in Section 3.4.1, together with the binary-encoded inputs (and intermediate signals, as needed) is then used to generate the final canonical TED representation.

Figure 3.15(b) illustrates this approach for a simple case of 2-bit operands, and demonstrates that both designs represent the same function, hence are equivalent. The 2-bit case is used here for illustration only, since the generated TED diagrams for larger would be too big to show here. However, the results shown in the previous section clearly demonstrate that this approach can be used to solve the equivalence verification problem for this design with up to (or even beyond) 256-bit operands.

3.4.3 Results

In this section, the TED representation was used simply to illustrate the concept of symbolic RTL verification rather than as a robust method to solve the equivalence verification problem. Nevertheless, TED, in addition to providing the word-level symbolic solution, can easily handle the Integer ALU design with up to 26-bit operands (beyond which the internal memory management is not efficient). The CPU runtime for such solutions is shown in Table 3.5. As we can see, the solution can be obtained in a matter of fractions of seconds. The experiments were run on a PC with Intel Processor Core i5-3470 CPU 3.20GHz x4 with 15.6 GB memory.

An alternative, and a more efficient solution is based on an approach that computes (i.e., extracts) the function performed by the design by rewriting the symbolic expressions of a design from the primary outputs to primary inputs. Such an approach has been used in our earlier work [32] in the context of arithmetic bit-level (ABL) networks, but applies verbatim here. In this approach, the specification polynomial (*input signature*) for a given design is computed from the word-level outputs (*output signature*) and expressed in terms of all the input variables: the operands A, B, C, control signals neg_i, and other configuration signals. Such computed signature is then

compared to the input signature obtained in a similar manner from the other design. The proof that such generated polynomials are identical is actually performed by running both designs in the same process and checking if $F_{Y1} - F_{Y2} = 0$. As shown in the Table (column *Function-Extract)*, this approach is highly scalable: it can solve the bit-level Integer ALU for operands with at least 256 bits in a matter of seconds.

In comparison, in[117] a commercial combinational RTL equivalence tool, Hector, was used to formally verify equivalence of a 16-bit instance of this ALU design. Solving this problem with Hector required 16-way case splitting (performed by hand) and solving the 16 simpler problems corresponding to some combinations of the configuration bits. The CPU time of 8 seconds reported in [117] cannot be used to compare to our results since the parameters of the computing platform were not given. Larger design were not attempted in there, claiming increased difficulty experienced by the solver.

Table 3.5: CPU time and memory results using TDS and Function Extraction

Op-size	TDS		Function-Extract	
	CPU (sec)	Mem (kB)	CPU (sec)	Mem(kB)
4	0.01	3400	0.01	2420
8	0.03	4470	0.03	4020
16	0.06	8904	0.11	9628
26	0.19	18160	0.32	21972
32	-	-	0.48	32280
64	-	-	1.93	124776
128	-	-	8.07	494304
256	-	-	34.66	1984464

3.5 Conclusions

This chapter presented the function extraction technique that derives the function computed by an integer arithmetic circuit from its gate-level implementation. It demonstrated that such function extraction and the test if the implementation satisfies the specification can be efficiently implemented in the algebraic domain using signature rewriting concept.

This approach uses an advanced data structure and a set of efficient heuristics to effect this extraction. The results show that the approach can handle gate-level integer multiplier circuits up to 512 bits and contain over 2 million gates. It should be noted that the experiments were conducted on circuits synthesized with ABC onto a relatively simple set of complex gates (*mcnc.genlib*). It seems that the synthesis tool which retains a certain degree of redundancy in the circuit, in the form of a *vanishing polynomial*, may be useful in verification. This type of redundancy is helpful in reducing the size of the intermediate polynomial expressions, which could significantly reduce the complexity of function extraction. Another observation is that solving the verification problem for highly optimized bit-level circuits, synthesized with commercial tools, remains a challenge. There are two possible reasons: 1) the synthesis tools are more aggressive in removing such redundancy; 2) the best ordering of rewriting is very difficult to identify, which causes memory explosion problem. This, together with the need to know the circuit I/O boundary is currently the main limitation of the method presented in this chapter.

We should also note that the verification of more structured circuits, containing larger pre-verified blocks will, in general, be easier. This is shown in the datapath verification case study. This is because it requires fewer polynomials to be processed, which lowers the overall size of the problem, and there are fewer rewriting iterations. This is especially true if the relationship between the inputs and outputs of such a block is simpler than those of the internal gates. The case study of datapath verification demonstrates that the high-level components significantly reduce the complexity of rewriting. The early work on verification of arithmetic circuits mapped into a combination of half- and full adders and logic gates demonstrate an almost linear computational complexity [7] [30]. However, as experimentally confirmed, sometimes the rewriting process benefits from breaking the aggregated complex gates into smaller ones to increase the chance of term cancellation during rewriting (Figure 3.5).

Based on those observations of the advantages and disadvantages of the function extraction technique, the next two chapters focus on improving its performance by introducing redundant polynomials (Chapter 4) and identifying the best ordering of rewriting for heavily optimized circuits (Chapter 5).

CHAPTER 4

COMPUTER ALGEBRA BASED VERIFICATION WITH REDUNDANT POLYNOMIALS

4.1 Introduction

Sequential circuits are composed of combinational arithmetic logic and memory components that are used to improve the throughput and energy efficiency of integrated circuits. In such circuits, the input is provided serially and the result is accumulated over a number of cycles to produce an n-bit (or word-level) result. The goal is to prove that the circuit computes the required arithmetic function collected sequentially at the primary outputs. Compared to combinational arithmetic circuits, the verification problem of sequential arithmetic circuits is much more complex, as it usually requires a complete exploration of the state space of the circuits. Even though functional verification of such arithmetic circuits can be cast as a combinational bounded model checking (BMC) problem, it is still challenging due to a large number of bits in practical arithmetic circuits. Boolean logic techniques, based on binary decision diagrams (BDDs) and satisfiability (SAT) solvers, have limited application to arithmetic circuits as they require flattening of the design into bit-level netlists. This chapter addresses the verification problem by modeling the sequential circuit as an *algebraic system* similar to that presented in Chapter 3, and proving that the polynomial word computed by the circuit matches the design specification, expressed in terms of the primary inputs. Specifically, it targets synchronous sequential arithmetic circuits with known required latency.

An example of the type of circuits considered here is shown in Figure 4.1. It is an n-bit serial adder built out of a single-bit adder, which operates for n clock cycles

to produce an $(n+1)$-bit result. An equivalent combinational model is obtained by unrolling the adder n times. The proof of functional correctness consists in transforming the polynomial associated with the result $Z = z_o + 2^1 z_1 + \cdots 2^n z_n$ into a polynomial expressed in primary inputs, $\{a_i\}, \{b_i\}$, applied to the circuit serially; and checking if this polynomial indeed represents the addition of two input operands: $Z = A + B = (a_o + 2^1 a_1 + \cdots 2^{n-1} a_{n-1}) + (b_o + 2^1 b_1 + \cdots 2^{n-1} b_{n-1})$. However, as demonstrated in this chapter, such a straightforward unrolling may be inefficient from the verification point of view, and special techniques are needed to make it effective and scalable. Those techniques are the main focus of this chapter.

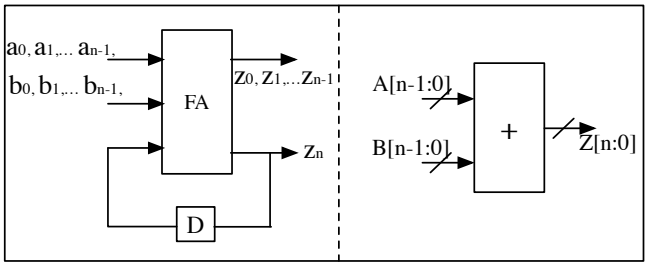


Figure 4.1: Sequential n-bit adder, $Z = A + B$.

4.2 Previous Work

A lot of research has been done in sequential equivalence checking, reachability analysis, state traversal, etc., applied to control logic, but relatively little has been published on *functional* verification of *arithmetic* circuits. Boolean satisfiability (SAT), which is an effective platform for encoding many CAD problems [14], [2], [53], has been used in verification of both control logic and arithmetic designs. SAT models for sequential designs typically rely on an Iterative Logic Array (ILA) representation by unrolling the combinational circuit component over a bounded number of cycles. Unfortunately, this technique when applied to modern industrial designs over a large number of cycles often exceeds the available memory resources [15].

A method for reducing sequential equivalence checking (SEC) of sequential logic into an equivalent combinational equivalence checking (CEC) is presented in [99]. That paper theoretically investigates when SEC can be reduced to CEC. It addresses the control part of large industrial designs, including pipelines but does not discuss the sequential arithmetic circuit verification. The work of [87] compares ATPG and SAT for checking safety properties and shows that, for relatively small circuits, two approaches are equally viable. Other analysis shows that sequential ATPG-based bounded model checkers outperform traditional SAT-based techniques, particularly for large designs [97].

4.3 Preliminaries

The functional verification method described in this chapter extends the combinational verification technique proposed in Chapter 3 to sequential arithmetic circuits. As we know that, it computes a unique bit-level polynomial function implemented by the circuit directly from its gate-level implementation. The difference between sequential and combinational arithmetic circuits is the signature. For combinational circuits, the output signature and input signature can be derived directly from the circuits using their binary encodings. However, for sequential circuits, the signatures are determined by the binary encoding of the bits, and the sequential behavior. To address this problem, the sequential arithmetic circuits are converted into combinational models by unrolling the circuit over k steps, where k is the required latency. Hence, the verification problem is converted into a combinational verification problem, which can be solved by rewriting the polynomial representing encoding of the primary outputs (the *output signature*) into a polynomial expressed in terms of the primary inputs (the *input signature*), using algebraic model of the internal gates. Note that the signatures derived from the unrolled model are not the sequential signatures

(introduced in next section) of the original implementations. The algebraic model is the same as shown in Chapter 3.

Input signature (sequential), denoted by Sig_{in}, is a pseudo-Boolean polynomial in primary input (PI) variables that uniquely represents an integer function computed by the circuit, i.e., its specification. Unlike the combination circuits, this input signature is derived based on the sequential behavior of the circuits. For example, input signature for a sequential adder shown in Figure 4.1 is Sig_{in} (at i cycle) = Sig_{in} (at $i-1$ cycle) + $(\sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i)$, where sum_{i-1} is the intermediate result at $i-1$ cycle.

Output signature (sequential), Sig_{out}, of the sequential circuit is defined as a pseudo-Boolean polynomial in the primary output (PO) signals in i cycles. Such a polynomial is uniquely determined by the binary encoding of the output. For example, the output signature of the serial adder shown in Figure 4.1 with output bits z_i is $Sig_{out} = \sum_{i=0}^{n-1} 2^i z_i$, where i is the clock step.

The proof of functional correctness is based on successively rewriting the output signature Sig_{out} into a signature in the primary inputs and comparing it with the expected input signature Sig_{in}. At each step of the procedure, an intermediate polynomial generated by the rewriting corresponds to some *cut* in the circuit, a set of signals separating primary inputs from primary outputs. The rewriting process recursively applies algebraic models of logic gates, followed by an algebraic simplification of polynomial terms to arrive at a unique algebraic expression. It also applies a *Boolean reduction* by reducing any occurrence of a nonlinear term x^k, to a single variable x. During rewriting of nonlinear terms, the size of an intermediate polynomial representing a cut in the circuit may increase exponentially, which seriously impacts the efficiency of the procedure. The size of the peak polynomial, commonly called the "fat belly", is a bottleneck for both the performance (CPU time) and memory used by the procedure.

The choice of the cuts (or, equivalently, the order in which the variables are eliminated by substitution) has big influence on the size of the fat belly and the efficiency of the rewriting process. A number of heuristics can be used to improve the efficiency, including: efficient data structure in the search of substituted variables; fast elimination of redundant terms; and other heuristics to minimize the size of the fat belly [31]. These techniques alone are not sufficient to avoid potential polynomial size explosion and additional techniques are needed. Some of them, specific to sequential arithmetic circuits, are discussed in the remainder of this chapter.

4.3.1 Vanishing Polynomials

Definition 1 *Vanishing Polynomial (VP): Starting with a Boolean signal v, the rewriting process generates a set of pseudo-Boolean polynomials* $\mathcal{P} = \{p_1, p_2, ..., p_i\}$ *(p_i is the polynomial when rewriting reaches PIs). If there is a subset* $\mathcal{P}'$=$\{p_1, p_2, ..., p_i\}$ *($2 < i < n$) such that each p is non-zero polynomial, and* p_{i+1}=p_{i+2}=...=p_n=0, *evaluated to* 0 *for all values, then* $\mathcal{P}'$ *are vanishing polynomials.*

Vanishing polynomials have been used to test if two fixed-size datapaths F_1, F_2 are equivalent by testing whether or not a difference polynomial, F_1-F_2, reduces to 0 over Z_2^m for the given bit-width [104]. Vanishing polynomials over Z_2^m have also been used as an optimization technique in high-level synthesis [45] by adding redundancy to the fixed bit-width polynomial computation in order to minimize the implementation. The vanishing polynomials in our work are similar to those used in high-level synthesis, but serve a different purpose. They are pseudo-Boolean expressions that always evaluate to 0 and insertion of such polynomials into the design will reduce the complexity of the verification process.

The use of vanishing polynomial in our work is illustrated with a 2-bit squarer circuit in Figure 4.2(a), with mathematical computation done by the circuit shown in Fig. 4.2(b).

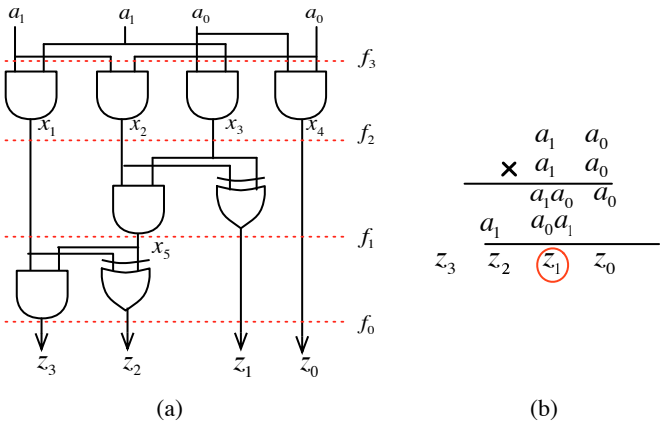


Figure 4.2: 2-bit combinational squarer circuit. a) Gate-level netlist; b) arithmetic squaring structure.

Close examination of the result shows that bit $z_1 = 0$, as it is a sum bit of two identical terms, a_1a_0 and a_0a_1. The resulting carry out bit (if any) is shifted one bit to the left and added to a_1 to produce z_2. Hence, with $z_1 = 0$, the initial starting point is $Sig_{out} = f_0 = 8z_3 + 4z_2 + z_0$ (without z_1). It is then transformed into f_1 using substitutions $z_3 = x_1x_5$ and $z_2 = x_1 + x_5 - 2x_1x_5$ (*c.f.* Eq. 7.1). Subsequent rewriting, using equation for $x_5 = x_2x_3$, results in $f_2 = 4x_1 + 4x_2x_3 + z_0$. It is then transformed (using equations for the AND gates), to produce $Sig_{in} = f_3 = 4a_1 + 4a_1a_0 + a_0$. Comparing this result with the expected specification, $F_{spec} = (2a_1 + a_0)^2 = 4a_1 + 4a_0a_1 + a_0$, shows that the circuit correctly computes the square function.

Now let us include the vanishing polynomial for z_1, which in terms of the immediate signal variables can be written as $z_1 = x_2 + x_3 - 2x_2x_3$. In the first step, f_0 is transformed into f_1 as before. Then, f_2 is obtained from f_1 using equations for x_5 and z_1, resulting in $f_2 = 4x_1 + 2(x_2 + x_3) + z_0$. Finally f_3 is obtained by substituting variables x_1, x_2, x_3, x_4 with the corresponding equations for AND gates in terms of the primary inputs, a_0, a_1. The result is the input signature $Sig_{in} = f_3 = 4a_1 + 4a_1a_0 + a_0$. It also demonstrates that this is a correct arithmetic function. However, note that

the intermediate form, $f_2 = 4x_1 + 2(x_2 + x_3) + z_0$, is simpler than the one computed without z_1, as it does not contain any nonlinear terms.

In general, vanishing polynomials contain terms that cancel other monomials during the cut rewriting and keep them smaller, especially for sequential arithmetic verification. This is because that many internal signals evaluate to zero iff the rewriting process reaches the PIs. These internal signals exist in the cascade arithmetic functions which are created by unrolling process. We demonstrate this using a Multiply-Accumulator (MAC) in Section IV-I in this chapter.

4.3.2 Don't-care Polynomials

Definition 2 *Don't-care Polynomial (DCP): Assuming d is a Boolean signal and $\mathcal{P}$=$\{p_1, p_2, ..., p_n\}$ is a set of polynomials of d which are generated by rewriting, if d is included in the arithmetic algorithm but excluded in the design, d and $\mathcal{P}$ are Don't-care Polynomials.*

Don't-cares are critical in verification because they can reduce the complexity of the netlist to be analyzed by equivalence checking. For example, in [133],[90] it is demonstrated that generating observability don't-cares for node merging, followed by SAT-based verification, greatly improves scalability and performance over other solutions. The don't-cares in our work are similar to those in [133], [90]. In our case, however, the role of don't-cares is to minimize the size of polynomials in each substitution step, rather than to minimize the computational complexity of SAT solving.

For example, let's assume that there is a 3-bit 2's complement adder, which the three LSBs $x[2:0]$ are used in the design. The MSB x_3 is the sign bit of the result of this addition. Based on Definition 2, x_3 can be used as a *don't care* polynomial. According to [31], the *output signature* should be $x_0 + 2x_1 + 4x_2$. However, we observe that the internal expressions explode in the rewriting process without *don't care* signal, x_3. We compare the internal expressions with/without x_3 in Figure

4.3. We can see that the peak size (i.e. the number of monomials) of the internal expressions without x_3 is 2× larger for a 3-bit adder. For larger designs, without the *don't cares*, the rewriting process will contain a 100x larger peak of internal expressions which causes memory explosion problem. Including this bit as part of the arithmetic algorithm can simplify the verification process because it contains potentially cancelable monomials. Similarly, we are able to verify a n-bit comparator by including the output bits $[n-2, 0]$ of a n-bit subtractor.

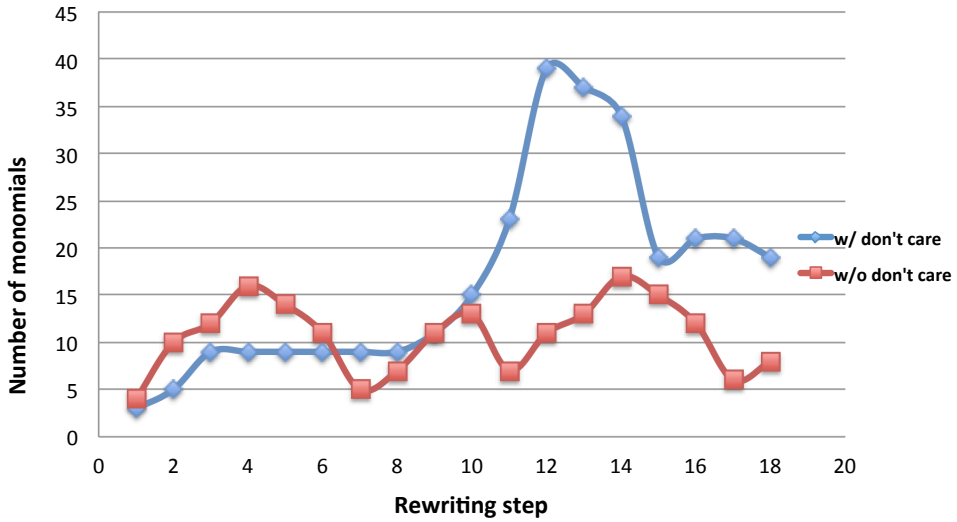


Figure 4.3: Compare the size of internal expressions with, without *don't care* polynomial x_3.

Another type of don't-care polynomials can be generated in unbounded sequential circuits. They can be obtained by expanding the reachable states to those that are not actually reached during the computation within the given range of input vectors [99] (in our case, within the given number of serial bits). A bit-serial squarer circuit, described in Section 4.2, will demonstrate this type of don't-care polynomial.

Including the *vanishing* and *don't-care* polynomials amounts to introducing redundancy into the original design, with the goal to improve the verification performance and scalability. This technique can be applied verbatim to formal verification of hard-

ware for cryptography applications, e.g., extension fields arithmetic circuits, used in *Advanced Encryption Standard* (AES).

4.4 Sequential Verification

In this section we show the application of VP and DCP to several types of sequential arithmetic circuits.

4.4.1 Multiply-Accumulator (MAC)

Consider an n-bit unsigned MAC circuit shown in Figure 4.4. The circuit should compute the result $R = \sum_{i=1}^{k} A_i B_i + C_0$ in k cycles, for k sets of n-bit inputs, $A_i[0...n-1], B_i[0...n-1]$, where $i = 1, ..., k$, and with some carry input vector C_0.

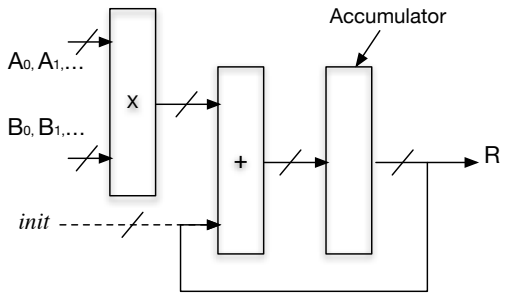


Figure 4.4: Original MAC circuit: $R = \sum_i A_i \cdot B_i + C_0$.

Figure 4.5 shows the unrolled version of the circuit for $n = 4$ bits and $k = 2$ cycles. The proof of functional correctness is obtained by transforming $Sig_{out} = \sum_{i=0}^{9} 2^i r_i$ using algebraic equations of internal gates of the circuit into an input signature, using the rewriting technique discussed earlier. The resulting input signature Sig_{in} is a function of the 4-bit primary inputs A_0, B_0, A_1, B_1 and C_0.

Note that the bit-widths of the two inputs to the first adder circuit in the unrolled model are different: they are 4-bit and 8-bit wide. Similarly, the bit-widths of the second adder are different (8 and 9 bits). This bit-width mismatch can be adjusted by the sign extension applied at the shorter inputs. Since the extension bits are all

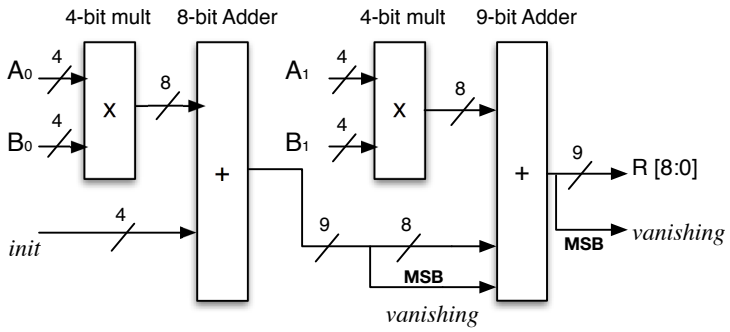


Figure 4.5: MAC circuit unrolled over two cycles.

"0", the most significant bit, i.e., the carry-out of the first adder, always evaluates to 0 (it is a *Vanishing Polynomial*). This, combined with the mismatch between the inputs to the second adder, cause the two MSBs of the output to evaluate to 0 as well. Therefore, it may seems logical to exclude the two most significant bits from the computation to simplify the model.

However, such a straightforward application of the signature rewriting scheme to this circuit is not efficient, as it may result in a large number of product terms of the intermediate signature before reaching the PIs. As a result, the CPU time and memory consumption can be prohibitive. For example, it takes more than 190 seconds of the CPU time and requires 2 GB memory to verify a 4-bit MAC circuit on our computing platform (see the Results section).

To simplify and speed up the verification procedure, we take advantage of the structure of the unrolled model by identifying the *Vanishing Polynomials* associated with the 0-function bits and adding them to the output signature Sig_{out}. As mentioned before, adding such a redundancy can simplify the elimination and substitution procedure during signature rewriting. Specifically, many cancellations will occur between the terms of the vanishing polynomial and other terms of the computed signature during the elimination and substitution process. After adding the vanishing polynomials, we could verify a 64-bit MAC in just 4 seconds, with only 142 MB mem-

ory – compared to 190 seconds and 2 GB memory for the 4-bit version of the circuit (see Tables 4.2 and 4.3).

4.4.2 Serial Squarer

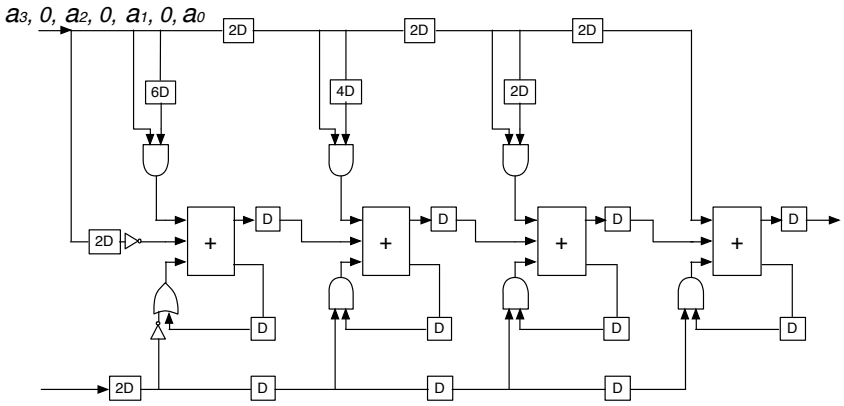


Figure 4.6: A 4-bit Serial Squarer.

Another type of redundancy encountered in bit-serial arithmetic circuits appears in a *serial squarer* circuit [37] which computes a square value of an n-bit integer input. An example of a 4-bit serial squarer is shown in Figure 4.6. The input bits of an integer number are provided serially over a single line, interleaved with 3 zeros, and outputs bits of the result are collected serially at the single output line. The proof of functional correctness is obtained by transforming the Sig_{out} using algebraic equations of internal gates of the circuit into an input signature. The delay elements D are modeled by unrolling each module $2n$ times. The fully unrolled model is too large to be shown in this section; a simplified model is shown in Figure 4.7.

To make the verification efficient, we extend bit-widths of the adder in the serial squarer circuit. Each of the four 1-bit adders is expanded over eight cycles, using standard techniques, into a combinational 8-bit adder. The resulting 8-bit adders as shown in Figure 4.7. The 8-bit input, B_2, to the second stage adder requires sign extension, while the other input is already 9-bit wide, as generated by the previous

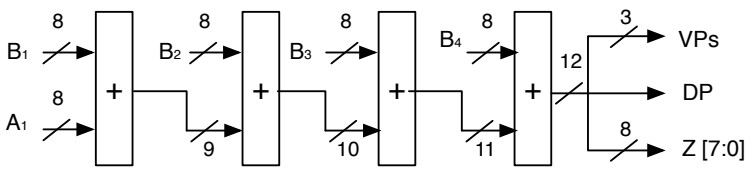


Figure 4.7: Unrolled 4-bit Serial Squarer.

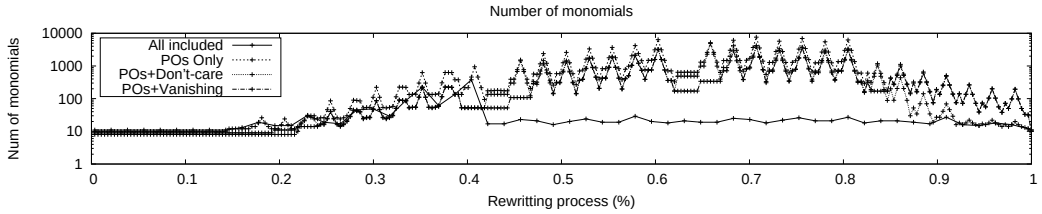


Figure 4.8: Evaluation of Don't Care and Vanishing polynomials on a 4-bit serial squarer.

adder. Similarly, the inputs to the remaining two adders, B_3 and B_4 are sign-extended by 2 and 3 bits, respectively. As a result, the 12-bit output is composed of extra four bits. The most significant three of those bits are always 0 because of the sign-extension; they can be represented by vanishing polynomials. The other bit however, is not a 0-functions, but a *Don't Care*, providing the two's complement corrector to the remaining 8-bit adder. This represents a reachable state that is unreached in the design.

By including both types of polynomials (vanishing and don't cares), the computation of the input signature can be greatly simplified. Specifically, it is used to simplify the algebraic equations during the rewriting process in order to minimize the size of the "fat belly".

4.5 Experimental Results

The sequential verification method described in this chapter has been implemented as a C++ program and tested on a number of sequential, serial arithmetic circuits, taken from [37] and [61]. The experiments were run on a PC with Intel Processor Core i5-3470 CPU 3.20GHz ×4 and 15.6 GB memory. Table 4.1 includes the CPU time and memory results for integer multiply-accumulator and add-shift multiplier circuits (an instance of the $s344/s349$ circuit from ISCAS-89 benchmarks, without the counter), extended to 256 bits.

Tables 4.2 and 4.3 summarizes the benefit of using vanishing polynomials and don't care polynomials in computing the input signature, and its effect on solving the sequential arithmetic verification problem. The table shows that using both, don't care and vanishing polynomials, gives best solution in CPU time and memory usage. The reason why using the don't care polynomials only is better than using the vanishing polynomials only is that the don't care bit, which is the first unreached state in the serial squarer design, contains more information and can produce more cancellation of intermediate terms.

Table 4.1: Verification results for GF(2^{256}) Adder, MAC, and Add-shift Multipliers

Circuit	256-bit		
	# Gates (Unrolled)	CPU [sec]	Mem
GF(2^{256}) Adder	3.5 K	0.21	1.1 MB
Add-Shift-Mult	587K	31.81	1.2 GB
2-Cycle MAC	1,049K	66.71	2.3 GB
6-Cycle MAC	3,148K	203.63	6.9 GB

To further analyze the effect of vanishing and don't care polynomials, we monitored the largest-size polynomial during rewriting (the fat-belly). The results are shown in Figure 4.8. The horizontal axis represents the time-line of the rewriting process as percentage of the complete run; the vertical axis represents the size of the expression at each point of the computation. We can see that the worst case

corresponds to the case when output signature contains only PO signals (without vanishing or don't care polynomials). Including both types of redundant polynomials significantly reduces the size of fat belly and of the largest monomial. In summary, including don't-care and vanishing polynomials can improve efficiency and scalability of arithmetic verification.

Table 4.2: Effect of Vanishing and Don't Care Polynomials for MAC (MO = Memory out of 8 GB)

n-bit	2-Cycle Integer MAC			
	Vanishing Poly		POs Only	
	CPU (sec)	Mem (MB)	CPU (sec)	Mem (MB)
4	0.01	2.2	190.86	2.1 GB
6	0.02	3.0	-	MO
8	0.06	3.9	-	MO
64	4.24	142.1	-	MO

Table 4.3: Effect of Vanishing and Don't Care Polynomials for Serial Squarer (MO = Memory out of 8 GB)

n-bit	Serial Squarer							
	Vanishing+Don't Cares		Don't Cares Only		Vanishing Only		POs Only	
	CPU (sec)	Mem (MB)	CPU (sec)	Mem (MB)	CPU	Mem (MB)	CPU	Mem (MB)
4	0.01	2.3	0.13	11.3	-	MO	-	MO
6	0.04	3.1	3.94	205.2	-	MO	-	MO
8	0.06	4.2	-	MO	-	MO	-	MO
64	4.71	161.8	-	MO	-	MO	-	MO

Table 4.4: Sequential Squarer results: comparison with SAT and SMT (TO = Time_out after 3600 sec)

Serial Squarer			Our		SAT [sec]				SMT [sec]		
Size	Clk Cycles	#.Gates	CPU [sec]	Mem	*minisat*	*ABC*	*lingeling*	*minisat_blbd*	*Boolector*	*Z3*	*CVC4*
4	11	255	0.01	2.32 MB	0.00	0.01	0.00	0.00	0.00	0.00	0.01
16	59	4.33K	0.26	11.7 MB	35.96	61.99	18.34	12.45	19.87	20.86	92.56
20	75	6.87K	0.42	17.3 MB	1698.53	TO	720.33	549.41	533.59	1045.18	TO
24	91	9.92K	0.62	24.1 MB	TO	TO	TO	TO	TO	TO	TO
64	251	71.2K	4.71	161.8 MB	TO	TO	TO	TO	TO	TO	TO
128	507	285K	18.63	667.3 MB	TO	TO	TO	TO	TO	TO	TO
256	1019	1.14M	77.12	2.63 GB	TO	TO	TO	TO	TO	TO	TO
512	2043	4.58M	331.64	10.5 GB	TO	TO	TO	TO	TO	TO	TO

Our verification method was also compared to SAT and SMT techniques for a bit-serial squarer circuit, ranging from 4 to 512 bits. The results, showing CPU runtime and memory usage, are given in Table 4.4.

SAT comparison: The functional verification problem was modeled as Boolean satisfiability (SAT) on a product circuit, generated with a miter, and solved using the ABC system [75]. A miter was created between the unrolled serial squarer circuit and the reference design (a multiplier with two inputs tied together), and the miter's output was tested for unSAT. Several SAT tools were tested, including ABC (*cec* command) [75], miniSAT [110] , *lingeling* [12], and *minisat_blbd*, the winners of 2014 SAT competition [98].

For SMT comparison, we tested Boolector 2.0.0 (first place in SMT Competition 2014) [82], as well as Z3, and CVC4 tools. We tried two models: 1) We directly translated the algebraic equations of the unrolled serial squarer into SMT2 format and modeled the specification (Sig_{out}-Sig_{in}) as a Pseudo-Boolean polynomial using Boolean vector operations. Among the SMT solvers, Boolector produced the best result for the serial squarer circuit, but it was only able to solve up to an 8-bit version of the circuit in 3,000 seconds of CPU time. 2) The product circuit (miter) was translated directly into SAT by converting the CNF model into SMT2 format. This approach showed better performance; it is the one shown in Table 4.4. As shown in the tables, our method has approximately linear CPU time complexity for all the tested circuits, and wins with the best SAT and SMT tools by several orders of magnitude of CPU times for designs above 16 bits.

4.6 Conclusions

In this chapter, two redundant polynomials are introduced to reduce the complexity of extracting the polynomial expressions from the gate-level implementations. This has been demonstrated with several case studies of functional verification of gate-level sequential arithmetic circuits. This method goes beyond simple unrolling to a sequential circuit into a combinational one, as it applies *vanishing polynomial (VP)* and *don't-care polynomial (DP)* that are specific to sequential circuit operation.

In addition to arithmetic verification, the proposed procedure can also be used to derive (i.e., extract) an arithmetic function implemented by the circuit by computing its input signature from the known output signature. It is shown that inserting a useful type of redundancy, the *VP* and *DP*, can greatly improve the verification time and scalability. One limitation of this approach is that, generation of *VP* or *DP* during unrolling requires certain account of domain knowledge of the circuit under verification, and is difficult to be automated. The next chapter will introduce how to identify the redundant polynomial automatically using *And-Inv-Graphs*.

CHAPTER 5

ADVANCED ALGEBRAIC REWRITING USING AND-INV-GRAPH

5.1 Introduction

Chapters 3 and 4 demonstrate that computer algebra techniques, which construct the polynomial representation of a gate-level arithmetic circuit, offer significant advantages for verifying arithmetic circuits. The main reason why computer algebra techniques can verify arithmetic circuits so efficiently is that it substantially reduces the polynomials by eliminating non-linear terms when constructing polynomials from the gate-level representation. For example, let a polynomial expression be $E = 2x_1 + a+b-2ab$, where x_1 is an output of an AND2 gate with inputs a and b. After rewriting the algebraic model of AND2$(a, b) = ab$, we have $E = 2ab + a + b - 2ab = a + b$. As a result, the non-linear term ab has been eliminated. However, it is shown that directly extracting the polynomial expressions of heavily optimized circuits is very difficult. This is because the intermediate polynomials can explode during rewriting. The main reason is that the ordering that provides a large number of non-linear monomial eliminations is difficult to be identified in synthesized and heavily optimized circuits. This is because the logic synthesis and technology mapping process destroy the original structure in order to minimize the delay and area cost. Note that non-linear terms could explode exponentially after rewriting the variables present in these terms if they are not eliminated at the right time, i.e., the rewriting is not properly ordered.

The order of rewriting or performing polynomial divisions has a significant impact on performance of the computer algebra techniques [100][125]. However, computer

algebra techniques may fail to find an efficient order of nodes in the gate-level arithmetic circuits. The main reason is that these techniques, such as function extraction presented in Chapter 3, are are restricted to the actually netlist. We compared the performance of algebraic methods on combinational gate-level multipliers when different reversed topological orders are used in Chapter 3. It shows that an efficient reversed topological order may not exist in the post-synthesized gate-level netlist. Even if such an order exists, it may be difficult to identify because complex standard cells obscure the possibility of reducing the polynomial under construction. In addition, redundant polynomials detected from combinational and sequential arithmetic circuits can provide significant polynomial reductions as shown in Chapter 4. However, detecting such polynomials is often impossible after the circuit has been restructured during automated or manual synthesis.

The approach presented in this chapter aims at improving the efficiency of algebraic rewriting in the context of arithmetic verification. It addresses the problem by using a compact and uniform representation of the Boolean network called the *And-Inverter Graph* (AIG) [77]. Instead of directly applying algebraic rewriting to the gate-level netlist, it is applied to an AIG, which offers a more functional view of the ordering of rewriting on the particular structure. Additionally, this approach allows for automatic handling of redundant polynomials, which significantly reduces the complexity of algebraic rewriting.

5.2 Background

5.2.1 Boolean Network

A Boolean network is a directed acyclic graph (DAG) with nodes representing logic gates and directed edges representing wires connecting the gates. And-Inverter Graph (AIG) is a combinational Boolean network composed of two-input AND-gates and inverters [77][63]. In an AIG, each node has at most two incoming edges. Each

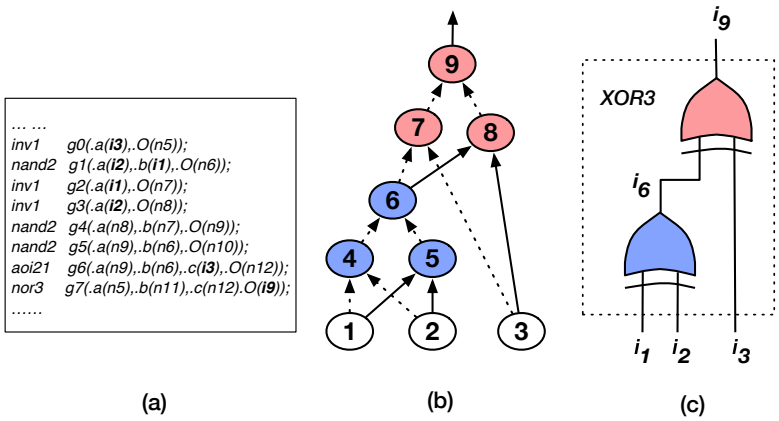


Figure 5.1: Representing circuits as AIGs. a) Post-synthesized XOR3 gate-level netlist. b) AIG of the synthesized XOR3 gate-level netlist. (c) The extracted two XOR2 functions (nodes 6 and 9) and one XOR3 function (node 9).

internal node in the AIG represents a two-input AND function. A node with no incoming edges is a primary input (PI). Primary outputs are represented using specific output nodes. Using DeMorgan's rule, the combinational logic of an arbitrary Boolean network can be transformed into an AIG [80], with the edges labeled properly to indicate the inversion of some signals. AIGs have been extensively used in logic synthesis, technology mapping [80] and formal verification [79].

AIGs have been used to detect unobserved Boolean functions such as *Multiplexers* [130][126] in an arbitrary gate-level circuits. This is done by computing a *Cut* in the AIG. A cut C of node n is a set of nodes of the network called *leaves*, such that each path from PIs to n passes through the leaf nodes. Node n is the *root* of a *Cut*. A *Cut* is K-feasible if the number of leaves does not exceed K. The *cut function* is the function of node n in terms of the cut leaves. An AIG node n in an AIG structure that represents a Boolean function F, is called an F-node. Each node is an AND function and the edges indicate the inversions of Boolean signals[1]. An example of

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

identifying XOR functions embedded in the AIG is shown in Figure 5.1. The AIG shown in Figure 5.1(b) represents a sub-circuit described in Figure 5.1(a). It includes a 3-feasible *Cut* of *node* 9 and a 2-feasible *Cut* of *node* 6, among other possible 3-feasible cuts. Let the function of an AIG node be i_x, and x be the index value of the node. The function of *node* 6 is $i_1 \oplus i_2$, and the function of *node* 9 is $i_1 \oplus i_2 \oplus i_3$. Hence, *node* 6 is an *XOR2*-node, and *node* 9 is an *XOR3*-node. This means that an embedded XOR3 function consisting of two XOR2s exists and can be detected in the sub-circuit shown in Figure 5.1(a). Similarly, an AIG can be applied to identify embedded *MAJ* functions.

$$\begin{aligned} &\neg a = 1 - a \\ &a \wedge b = ab \\ &MAJ3(a,b,c) = ab + ac + bc - 2abc \\ &XOR3(a,b,c) = a \oplus b \oplus c = a + b + c - 2ab - 2ac - 2bc + 4abc \end{aligned} \tag{5.1}$$

5.2.2 Simplified Polynomial Construction

According to Chapter 3, efficiency of algebraic rewriting of Sig_{out} is determined by the amount of simplification during polynomial construction. This is because there is a large number of non-linear terms generated by *carry-out* (MAJ) and *sum* (XOR) functions, since the multiplication is performed by a series of additions. Finding the maximum polynomial cancellations has been previously addressed by improving the topological order of the gates [125]. For example, let a sub-polynomial expression be Nx_1+$2Nx_2$+..., where x_1 = *XOR3(a, b, c)*, x_2 = *MAJ3(a, b, c)*, where *a, b, c* are the inputs of XOR3 and MAJ3 functions. According to Equation 5.1, rewriting x_1 and x_2 together, four non-linear terms, namely $2Nab$, $2Nbc$, $2Nac$ and $4Nabc$, generated by the algebraic models of XOR3 and MAJ3 are eliminated. However, if rewriting is applied directly to the gate-level netlist, its efficiency is restricted when

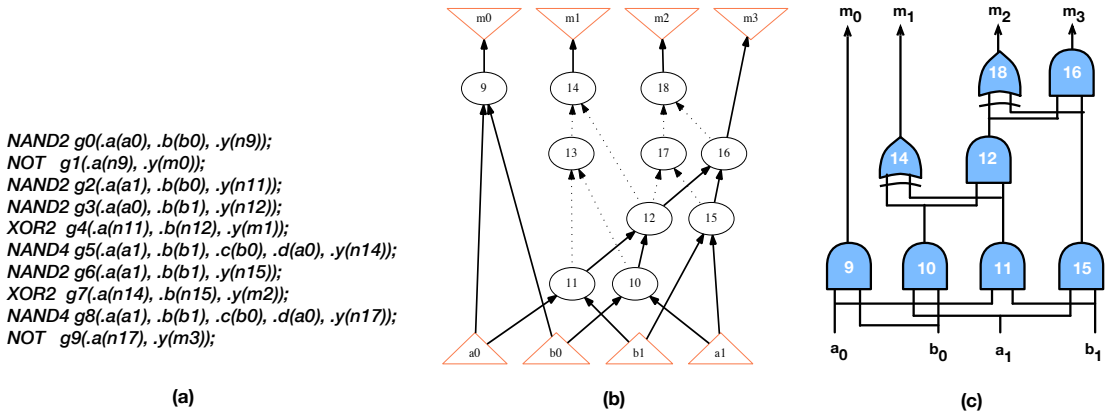


Figure 5.2: (a) AIG representation of a post-synthesized 2-bit multiplier gate-level netlist; (b) The AIG of the 2-bit multiplier shown in Figure 5.2(a); (c) Detected unobserved functions from the AIG and the correspondences to AIG nodes.

the MAJ3 and XOR3 functions are mapped into other standard cells by logic synthesis and technology mapping. For example, the XOR3 function mapped using standard cells is shown in Figure 5.1(a). In this case, there is no ordering that provides the maximum polynomial reductions.

5.3 Approach

This section presents the algebraic rewriting approach based on AIGs. Similarly to the approach presented in Chapter 3, the algebraic rewriting process rewrites the output signature for all AIG nodes in a reversed topological order. As discussed in Section III-E, the rewriting order that provides a large number of polynomial reductions, has significant impact on the performance of rewriting. However, there are many reversed topological orders available in an AIG, since many nodes can have the same topological depth. This approach detects a reversed topological order for algebraic rewriting that provides the maximum polynomial reduction. This is achieved by detecting pairs of MAJ3 and XOR3 nodes using AIG-based *cut enumeration*, and rewriting across the entire *MAJ3* and *XOR3* functions.

Algorithm 2 Algebraic Rewriting in AIG

Input: Gate-level netlist, output signature Sig_{out}
Output: *Pseudo-Boolean* expression extracted by rewriting

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

5.3.1 Outline of the Approach

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

- **Step 1:** converting the gate-level implementation into AIG.
- **Step 2:** detecting all pairs of XOR3 and MAJ3 nodes with identical inputs; topological sorting the AIG nodes while considering the detected pairs as one element. The second step is performed as follows:
  - Computing all 3-feasible (3-input) cuts of all AIG nodes.
  - Computing truth tables of all cuts.
  - Storing cuts in the hash table by their ordered set of inputs.

- Detecting pairs of 3-input cuts with identical inputs belonging to different nodes, such that the Boolean functions of the two cuts with the shared inputs belong to the *NPN* classes of XOR3 and MAJ3, respectively.

- **Step 3:** applying algebraic rewriting from POs to PIs following the reversed topological order determined in step 2). Note that XOR2 and MAJ2(AND2) are the special cases of XOR3 and MAJ3, where one of the inputs is constant zero.

Note that, in this approach, matching the XOR3 and MAJ3 nodes does not require the inputs and outputs polarity to be the same. Instead, all the cut-points are matched without considering their complemented attributes. For example, instead of being an exact XOR3, the function of a 3-feasible cut can be either XOR3 or XNOR3. Similarly, instead of being exactly MAJ3, the MAJ3 function can be one of the eight functions forming the *NPN* class of MAJ3 [51]. To compute the cuts, the 3-input cut enumeration is performed in a topological order as described in [86]. The truth tables of the cuts are obtained as a by-product of the cut enumeration. Thus, when two fanin cuts are merged during the cut computation and the resulting cut is 3-feasible, the truth tables of fanin cuts are permuted to match the fanin order of the resulting cut. These truth tables are then ANDed or XORed, depending on the node type, to get the resulting truth table. For the case of 3-input cuts, a dedicated pre-computation reduces the runtime of truth table computation to a small fraction of that of cut enumeration.

As soon as the XOR3 and MAJ3 pairs are detected, algebraic rewriting will be applied to the AIG network in a constrained reversed topological order, in which each XOR3 and MAJ3 pair is considered as one element. This means that at one topological depth, whenever either XOR3 or MAJ3 node of a pair (or its complement) is rewritten, the node of the other type is subsequently rewritten. The AIG nodes with the same topological depth that do not belong to any pair are ordered in the

decreasing order of their integer IDs. The algebraic rewriting ends when all elements in AIG network have been rewritten. The algorithm returns the extracted input signature.

Example 1 (2-bit CSA-multiplier): The mapped gate-level netlist of a 2-bit CSA-multiplier is shown in Figure 5.2(a). First, the gate-level netlist is converted to an AIG (Figure 5.2(b)). Next, a set of XOR3 nodes X, and a set of MAJ3 nodes M are detected: $X = \{14, 18\}$, $M = \{12, 16\}$. *Node 14* is *XOR3*(*10, 11, 1'b0*) and *node 12* is *MAJ3*(*10, 11, 1'b0*), where *node 10, node 11* and constant zero *(1'b0)* are the inputs; *node 18* is *XOR3*(*12, 15, 1'b0*) and *node 16* is *MAJ3*(*12, 15, 1'b0*); *1'b0* denotes Boolean *false*. Hence, two pairs of XOR3 and MAJ3 are generated, (*14, 12*) and (*18, 16*). The order of rewriting is determined as follows: 1) *node 18* is the node with highest depth; it is detected as XOR3 and paired with MAJ3 *node 16*; hence, the first rewriting starts from node 18 and 16, and ends at node 12 and 15; 2) similarly to the first rewriting, the second rewriting starts from nodes 14 and 12, and ends at nodes 11 and 10; 3) the remaining AIG nodes are ordered by their index value in decreasing order. The logic network after detecting all XOR3 and MAJ3 node is shown in Figure 5.2(c).

5.3.2 Detecting Redundant Polynomials

Significant simplification of polynomial construction can be achieved not only by performing algebraic rewriting using a reversed topological order, as discussed above, but also by detecting redundant polynomials, such as *don't-care* polynomials and *vanishing* polynomials [100][127]. Vanishing polynomials are those that always evaluate to zero; vanishing monomials used in the work of [100] are examples of such polynomials. Don't care polynomials can be identified in circuits (such as multipliers) with truncated outputs. Arithmetic operators are often truncated to reduce power consumption or speed up the critical path. The removed signals in those circuits contain

algebraic information needed to cancel algebraic terms of the remaining output bits. Polynomial associated with the most significant bit (MSB) of an adder or a multiplier is an example of such a polynomial.

To efficiently apply algebraic rewriting to the multipliers with output bits truncated, an approach that generates *don't-care* polynomials is presented. This approach is based on an observation that the logic obtained by removing output bits is either a carry-out function or a sum function of a full adder. It is known that MAJ3 and XOR3 with the same inputs are the components of a full adder. Hence, using the approach of detecting pairs of XOR3 and MAJ3, the XOR3 and MAJ3 nodes that do not belong to any such pairs are also identified. For example, a n-bit CSA-multiplier with *2n-1* output bits (with MSB removed), there is a missing MAJ3, i.e., the MAJ3 nodes with identical inputs of an unpaired XOR3. Since one pair of XOR3 and MAJ3 is a full adder, removing the carry bit (MAJ3) makes the function an addition *modulo 2*. In this case, the algebraic model of XOR3 (Equation 5.1) is reduced to $a \oplus b \oplus c = a+b+c \ mod\ 2$.

Example 2 (3-bit CSA-multiplier with MSB z_5 deleted): The AIG after detecting XOR3 and MAJ3 pairs of a 3-bit post-synthesized CSA-multiplier with MSB deleted is shown in Figure 5.3. The detected {XOR3 and MAJ3} pairs are represented using the ID of the root node of the XOR3 and MAJ3 nodes. We can see that there is one XOR3 (composed of two XOR2 nodes, *41* and *44*) with inputs $i_{36,37}$, $i_{27,29}$ and i_{38}, that cannot be paired with any MAJ3. This is because synthesis process removed the redundant logic (last carry out) when the MSB has been removed. In this case, the algebraic model of that XOR3 is reduced to $2^4 \cdot z_4(i_{49}) = 2^4 \cdot (i_{36,37} + i_{27,29} + i_{38})$.

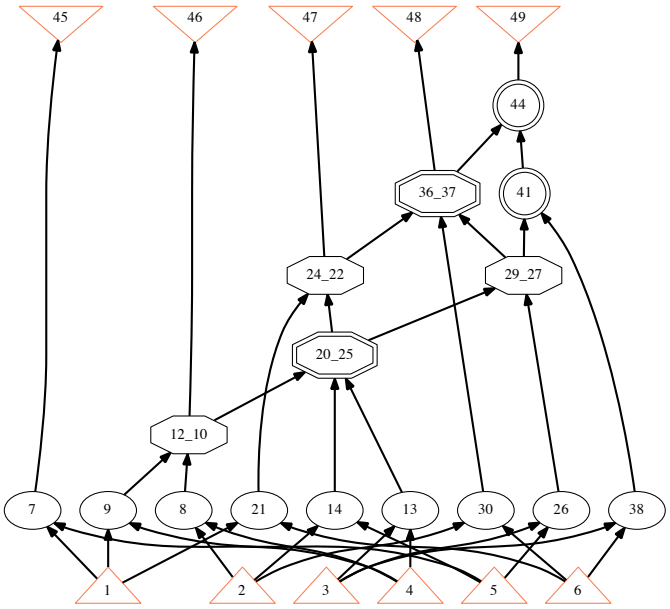


Figure 5.3: Detecting {*MAJ3-XOR3*} pairs of a 3-bit post-synthesized CSA-multiplier with MSB z_5 deleted.

5.4 Results

The technique described in this paper has been implemented in ABC [80]. It applies algebraic rewriting to the AIG and generates the polynomial input signature. The experiments include of Carry-Save-Adder (CSA) multipliers up to 512 bits. The results are compared with *functional extraction* presented in Chapter 3. The results show that the proposed technique is more efficient than the state-of-the-art technique for extracting the polynomial expressions for the CSA multipliers. The experiments were conducted on a PC with Intel(R) Xeon CPU E5-2420 v2 2.20 GHz x12 with 32 GB memory.

Evaluation of the proposed AIG-based algebraic rewriting for pre-synthesized and post-synthesized CSA multipliers is shown in Table 6.1. The same results for post-synthesized complex unsigned arithmetic circuits are conducted with the same benchmarks used in Chapter 3 are shown in Table 6.2. The runtime and memory usage

Table 5.1: Results of applying AIG-based algebraic rewriting to pre- and post-synthesized CSA multipliers compared to *functional extraction* presented in Chapter 3. **t(s)* is the runtime in seconds. **mem* is the memory usage in mb.

#bits	*Pre-synthesized*				*Post-synthesized*			
	Function extraction		This approach		*Function extraction*		This approach	
	t(s)	mem	t(s)	mem	t(s)	mem	t(s)	mem
64	1.89	74	0.04	34	5.50	76	0.04	34
128	8.12	288	0.15	117	39.64	299	0.16	120
256	32.65	1157	0.82	441	285.22	1250	0.82	439
512^2	130.22	4427	3.76	1695	-	-	-	-

Table 5.2: Results of applying AIG-based algebraic rewriting to post-synthesized complex arithmetic circuits compared to *functional extraction* presented in Chapter 3. **MO* = Memory out of 8 GB.

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

are compared to *functional extraction*. In Table 6.2, the functions of the arithmetic circuits are shown in the first column.

We can see that the runtime of the proposed approach is less than a second for both the pre- and post-synthesized CSA multipliers of any bit-width. The memory usage has been reduced on average by 60%, compared to *functional extraction*. Note that the complexity of extracting polynomial expressions using functional extraction is increased when the multipliers are synthesized. For example, extracting post-synthesized 256-bit multiplier using functional extraction requires 9× more runtime and more memory. However, using the proposed AIG-based approach, the runtimes of extracting pre- or post-synthesized multipliers are almost the same. More importantly, we can see that our approach outperforms *functional extraction* on complex arithmetic circuits (Table 6.2).

5.5 Conclusion

In this chapter, a method of significantly improving the efficiency of algebraic rewriting used in arithmetic verification, has been proposed. The method is based on the AIG representation of the Boolean network. This approach can formally verify practical multipliers that are heavily optimized and mapped using 14nm CMOS technology library. Additionally, we introduce a technique to automatically handling redundant polynomials. At this point, the function extraction technique is well tuned and is shown to be applicable to large industrial designs.

CHAPTER 6

ALGEBRAIC SPECTRUM - A NEW CANONICAL REPRESENTATION OF ARITHMETIC

6.1 Introduction

With an ever increasing complexity of integrated circuits and systems on chip contemporary designs are built from predesigned modules and IP blocks. While in principle such blocks are pre-verified and should be secure, they may come from untrusted sources and require additional verification. Furthermore, those circuits are often modified, resynthesized and retimed for one of several reasons: i) added security (using elaborate obfuscation techniques); or ii) to further bit-optimize the design in the context of the surrounding logic. As a result the hierarchy structure is destroyed and high-level module information is lost, resulting in increased fanouts and sharing between logic. The verification of such resynthesized circuits poses a serious challenge to the verification problem. This problem is particularly acute for arithmetic circuits and datapaths, which by definition are characterized by large bit-widths and cannot be efficiently solved using Boolean methods. Many high-level verification techniques have been developed for high-level descriptions, such as Register transfer level (RTL), or system-level, using *SystemC*, *SystemVerilog* description languages. However, as argued above, hardware verification still needs to be done on low-level design representations with gate-level netlists. Boolean logic techniques based on Binary Decision Diagrams (BDDs) or Binary Moment Diagrams (BMDs) and satisfiability (SAT) solvers cannot handle complex arithmetic designs flattened to bit-level netlists on which these tools operate.

One possible (and in our view, most promising) way to solve this verification problem is to abstract the low-level design to a higher-level representation at which one can reliably and efficiently reason about the underlying design. This offers insight into the internal structures of the designs, which is important from the security point of view; it enables a more comprehensive analysis of the circuit that can be used for understanding the actual functioning of the (possibly maliciously modified circuit), or for adding security (creating obfuscation). Again, this problem is particularly challenging for arithmetic circuits and datapaths due to the inherent high bit-level complexity. Abstracting the word-level information from gate-level netlists, while maintaining the useful information about the control and connecting logic is a well known method that enables the high-level formulation. This approach, typically referred to as *reverse engineering* [66], can provide significant improvement in performance and scalability. However, the work on abstracting the designs from gate-level implementations is rather scarce (to be reviewed in the Related Work section).

In summary, abstracting word-level information from gate-level designs is important for several reasons: a) In *formal verification* of complex arithmetic designs by bringing the design representation to a higher level in order to simplify the verification process; b) In *reverse engineering*, when the higher level model of the design (such as RTL) is not available or when the circuit was build bottom-up; c) In *hardware trust* and security applications, where the circuit needs to be analyzed to properly isolate maliciously inserted hardware; or d) To understand the *general structure* of the design. Identifying higher-level blocks can also be viewed as a *generalized technology mapping* by grouping lower-level components into higher level blocks from the library of complex arithmetic operators.

This chapter addresses the abstraction problem for arithmetic circuits and datapaths. In particular, it presents a systematic way to abstract word-level structures from gate-level implementations of combinational integer arithmetic circuits. The

proposed method represents the gate-level circuit in algebraic domain by representing the circuit components as pseudo-Boolean polynomials; it then uses a recently developed polynomial rewriting technique to extract arithmetic function(s) embedded in the circuit. During the rewriting the intermediate pseudo-Boolean expressions are examined in order to identify possible word-level structures. The identification is done using a novel *spectral analysis* technique, which matches the parsed polynomial expressions against the reference *spectra* of basic arithmetic blocks, such as multipliers, adders, and multiply-and-accumulate operators. Each arithmetic operator has its own, unique spectrum representation, which helps in identifying its presence in the design, regardless of its internal representation. The proposed approach is able to abstract the word components from polynomial expressions and reason about the word-level structure from the internal expressions. By representing logic and arithmetic functions as pseudo-Boolean polynomials, it is possible to mitigate the size explosion typically encountered in Boolean domain.

Specifically, the abstraction problem is solved in three steps:

1. Parse the expressions to identify the word candidates during the polynomial rewriting process;

2. Classify the word candidates as linear and non-linear components and extract the word information by lexicographical (structural) matching of the terms.

3. Generate the correspondence between the words and gate-level netlist and use it to reason about the word-level arithmetic operations.

6.2 Related Work

One of the most successful and cited abstraction techniques is Counterexample-Guided Abstraction Refinement (CEGAR) [35]. It has been shown to be an effective paradigm in a variety of hardware and software verification scenarios. Clarke et. al.

[35] successfully demonstrated how to automate abstraction and refinement in the context of model checking for safety properties of hardware and software systems. In particular, these approaches create a smaller abstract transition system from the underlying concrete transition system and iteratively refine it with the spurious counterexamples produced by the model checker. Additional CEGAR approaches based on the extraction of unsatisfiability explanations derived from the unfeasible counterexamples that provides stronger refinement of the abstract model and significantly reduce the number of refinement iterations [52][3]. However, all these works target model checking with a given property, instead of extracting the actual function. Most importantly, they are only applicable to bit-vector behavioral RTL implementations. Andraus et. al. [4] describe a methodology for datapath abstraction that is particularly suited for equivalence checking. In their approach, datapath components are automatically abstracted to uninterpreted functions using the ACL2 theorem prover. However, this work also considers only behavioral Verilog models.

Recently some interesting research has been done in programming (software) abstraction, including verification of out-of-order microprocessors, *term-level* abstraction useful in microprocessor design verification, term-level bounded model checking, correspondence checking, refinement verification, predicate abstraction, etc. [73][52][50]. However, these techniques apply to bit-vector behavioral RTL and software abstraction and are not discussed here as not relevant to our work.

Abstracting the word-level information from gate-level netlists is often referred in literature as *reverse engineering*. One of the first works on reverse engineering is credited to Hansen et al. [49]; it presents several strategies of reverse engineering circuit functionality from a gate-level schematics using ISCAS-85 combinational circuits. The proposed methods are mostly manual and include searching for common library components and repetitive structures. They relay on canonical truth tables of

smaller blocks but can also identify some bus structures and control signals. However, they do not formally characterize the abstraction problem.

Torrance et al. [118] describe the practice of reverse engineering of the manufactured products, including product tear-downs, identifying components on a board and performing functional analysis through probing, eventually deriving a schematic from a stripped IC. Our work starts at this point, assuming knowledge of gate-level schematic.

In [113], the authors propose techniques to identify high-level components such as register files, counters, adders and subtractors.

In [67], the authors formalize the problem of reverse engineering as mining temporal properties and graph matching against the logical specification. This method uses *pattern mining*, a technique of mining interesting behavioral patterns from the simulation or execution traces applied to a gate-level netlist. The components of the library and the behavioral patterns are represented as pattern graphs. The input-output signal correspondences of the abstract components are found by matching sub-circuits against a library of abstract components using subgraph isomorphism between the pattern graphs. The maximum common subgraph (MCS) between the pattern graphs defines candidate signal correspondence mappings. The function of the sub-circuit is then determined by finding the closest match in the component library. This step is similar to the traditional DAG-based technology mapping technique [57] used in logic synthesis. It is followed by formal model checking, to verify correctness of the matching of a given extracted component against each logical specification. The matched sub-circuits are combined to generate the final high-level description.

A more comprehensive approach, described in [66], is based on analyzing an unstructured netlist as opposed to effecting sub-circuit matching.It is not based on simulation traces and instead models the problem in a combination of functional and

structural domain. Together with [113], this work is probably the most relevant to ours. The authors present a variety of techniques to identify high-level components, such as adders and subtractors, applicable to arithmetic datapath extraction. The first stage identifies candidate words using two complementary techniques: structural shape hashing and functional bitslice aggregations. *Shape hashing* represents the backward reachable gates in a feasible depth from a given wire, while the *bitslice aggregation* technique additionally finds similar wires by functional matching and groups equivalent wires into words. They also addressed the reverse engineering problem in the context of extensive logic sharing in an optimized flattened netlist, by solving the problem using Quantified Boolean Formula (QBF). The next stage infers more words by iteratively propagating candidate words across gates in the netlist using symbolic evaluation. The final stage looks for word-level operations, such as addition and rotation. It is simply done by "cutting out the portion of the netlist that lies between words", and then check if this structure implements a particular word operation. However, this technique is not efficient for large non-linear arithmetic operations since it requires extensive *bit-blasting*. Additionally, for long cascaded word operation such as Multiply-Accumulator (MAC), it requires many *word propagate* iterations that limit its applicability. Our work aims at overcoming these limitations.

A recent reverse engineering techniques, proposed in [105], relies on simulation to identify candidates for a given arithmetic operation. First, it identifies a set of candidate blocks that contain the expected word-level operations. Then it finds the word-level blocks among the candidates that match some components in a given library. A set of permutation-invariant simulation vectors are used to construct the simulation-graphs (SGs), and the matching problem is solved by subgraph isomorphism. However, this technique is not applicable to a design that contains serial arithmetic operations, such as multiply-and-accumulate (MAC). To demonstrate this point, we tested a simple 2-bit MAC design $F = A \cdot B + C$ using the tool provided

in the paper [105]. This design includes one 2x2-bit multiplier which feeds into a 4-bit adder. The goal was to search for a 2x2-bit multiplier in the design. The adder and multiplier are generated using ABC tool and the entire design is synthesized and mapped by ABC [74]. The given library component contains a 2x2-bit multiplier and a 4-bit adder. Then, we tested that if the tool is able to identify if there is a 2-bit multiplier. Unfortunately, the tool was unable to identify the multiplier, always returning the prompt "no pattern contained in the target" with multiple options. Another limitation of this work is that it is only able to identify the multiplier up to 8-bit wide. As demonstrated in the Results section of the proposal, our method can identify the presence of both operators (multiplier and the adder) up to 128-bits, using our novel and original technique of *spectral analysis*. The spectrum of each operator is obtained at the end of the process, indicating that this is a MAC.

6.3 Algebraic Spectrum

This section introduces a novel concept of *algebraic spectrum*. We prove its uniqueness, and illustrate its application to arithmetic combinational equivalence checking problem, and word-level abstraction.

Given an algebraic polynomial expression P of function F, each monomial that has k variables is called *k-variable* monomial, or *k-var* monomial for short, where $1 \leq k \leq n$, and n is the total number of variables in F. We define $\mathbb{M}_k$ as an ordered set of *k-var* monomials $\{m_i\}$ in increasing order of their coefficient value c_i. $\mathbb{C}_k$ is the set of coefficients of $\{m_i \in \mathbb{M}_k\}$ with the same order as $\mathbb{M}_k$. Let $N_k(c_i)$ be the number of *k-var* monomials of P with coefficient c_i.

Definition 1 (Algebraic Spectrum): A *k-var* spectrum is defined as a vector of integers, $S_k(\mathrm{P})=\{N_k(c_i), \forall\ c_i \in \mathbb{C}_k\}$, ordered by increasing value of c_i.

Example 2: Given a polynomial expression $P=x + 2z + 4xy + 4xz + 5xyz$. We have

$$\mathbb{M}_1 = \{x, 2z\}; \mathbb{C}_1 = \{1, 2\}, S_1 = \{1, 1\}$$

$$\mathbb{M}_2 = \{4xy, 4xz\}; \mathbb{C}_2 = \{4, 4\}, S_2 = \{2\}$$

$$\mathbb{M}_3 = \{5xyz\}; \mathbb{C}_3 = \{5\}, S_3 = \{1\}$$

Note that the sum of all *k-var* spectra represent the entire polynomial expression. In this example, algebraic spectrum of P is S_1+S_2+S_3.

6.3.1 Uniqueness of Algebraic Spectrum

Theorem 1 (Uniqueness): An arithmetic function composed of basic operations of addition and/or multiplication has a unique algebraic spectrum.

Proof: The proof is based on the fact that such arithmetic functions have unique polynomial representation, where each monomial is a single variable or a product of variables and coefficients are integer. It then suffices to show that such a polynomial has unique algebraic spectrum. While this is not true for arbitrary polynomial, we will demonstrate that this holds true for arithmetic functions involving addition and multiplication. The proof is achieved in four basic steps:

- Prove that the spectrum of additions with m operands is always a linear 1-*var* spectrum.
- Prove that there is a unique spectrum for multiplication with two operands.
- Prove that there is a unique spectrum of multiplication with m operands.
- Since the sum of all *k-var* spectra represents the function, if statements 2 and 3 hold, the spectrum of any combination of addition and multiplication is unique.

Addition with m operands: An m-operand n-bit arithmetic addition is the sum of a set of bit-vector words, W_1+W_1+...W_m. Polynomial expression of each word is $W_i = w_i^0 + 2^1 w_i^1 + \cdots + 2^{n-1} w_i^{n-1}$, where n is the bit-width of word W_i, and w_i^j is the

bit of word W_i at j^{th} position with coefficient 2^j. Hence, the polynomial expression of arithmetic addition is:

$$P_{Add} = \sum_{i=1}^{m} \sum_{j=0}^{n-1} 2^j w_i^j \tag{6.1}$$

We can see that all the monomials in P_{Add} are 1-*var* monomials. Additionally, we can see that the number of monomials with coefficients 2^j, j=$\{0,1,2,...n-1\}$, is always m. That is, for an m-operand n-bit arithmetic addition, $N_1(c_i)$=$\{m, \forall c_i \in \mathbb{C}_1\}$. Hence, the spectrum of this arithmetic addition S=S_1=$\{m,m,\cdots,\text{m}\}$ (m repeated n times), is unique.

Multiplication with 2 operands: Similarly, the polynomial expression of a 2-operand multiplication $A \times B$ is the product of polynomial expressions of A and B, where $A = \sum_{i=0}^{n-1} 2a_i$, $B = \sum_{i=0}^{n-1} 2b_i$, and n is the bit-width. The polynomial expression P_{Mult} of $F = A \times B$ is:

$$P_{Mult} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} a_i b_j \tag{6.2}$$

- Each monomial of P_{Mult} is a 2-*var* monomial, so the spectrum is composed entirely of S_2.
- Note that the coefficients c_i in $\mathbb{C}_2$ are generated by multiplying all pairs of coefficients of expressions A, and B, i.e., of two identical sets of integers, $\{2^0, 2^1, 2^2, ..., 2^{n-1)}\}$. Hence, $\mathbb{C}_2$=$\{2^0, 2^{1+0}, 2^{0+1}, ..., 2^{i+j}, ..., 2^{n-1+n-1}\}$, where $i, j \in [0, n-1]$.
- The number of 2-*var* monomials $a_i b_j$ for $i, j \in [0, n\text{-}1]$, is fixed and unique, but some of them may have the same coefficients.

- The number of unique coefficients, $N_2(k)$ with value k=2^{i+j} for $i, j \in [0, n\text{-}1]$, is also fixed and unique. This can be represented by Equation 6.3. Hence, the spectrum of 2-operand multiplication is unique.

Examples of 2-operand multiplication spectra with $n = \{2, 3, 4, 5\}$ are shown in Figure 6.1.

$$S_2 = \begin{cases} c+1 & 0 \leq c \leq 2^{n-1} \\ (2n-1)-c & 2^{n-1} < c \leq 2^{2n-2} \end{cases} \tag{6.3}$$

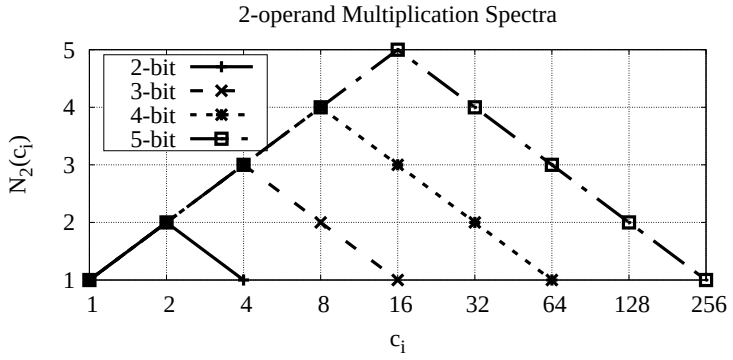
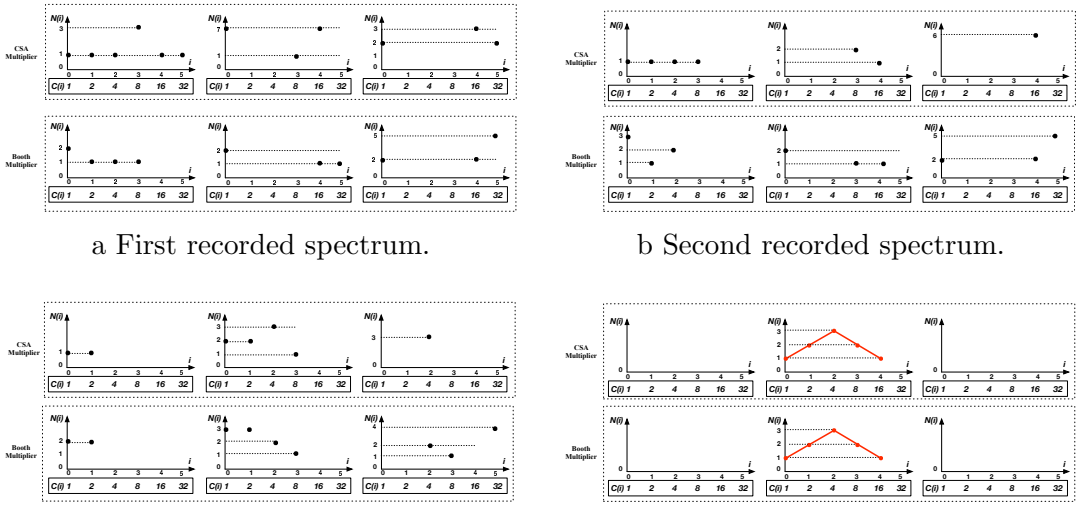


Figure 6.1: The spectra of 2-bit, 3-bit, 4-bit and 5-bit two-operand multiplication.

Multiplication with m operands: For an n-bit multiplication with m operands, the coefficients are generated in the same way as in two-operand multiplication, but cascaded in m-1 levels. The polynomial expression of an m-operand multiplications $P^m_{Mult} = A_1 \times A_2 \times ...A_m$ can be represented using Equation 6.4. Each word A_i ($i = \{1, 2, ..., m\}$) is an n-bit word. Each term a^j_i is the bit in word A_i at j^{th} position. Based on Equation 6.4, we can see that all the monomials in P^m_{Mult} are m-var monomials. This means that the spectrum of an m-operand, n-bit multiplication is S=S_m. Then, $\mathbb{C}_m$=$\{2^0, 2^1, 2^1, ..., 2^{i_1+i_1+...+i_m}, ..., 2^{m\cdot(n-1)}\}$, for $i \in$ {0,1,2,...,n-1}. We can see that



a First recorded spectrum. b Second recorded spectrum.

c Third recorded spectrum. d Final recorded spectrum.

Figure 6.2: Spectra of a 3-bit Booth-multiplier and CSA-multiplier of the four recorded expressions.

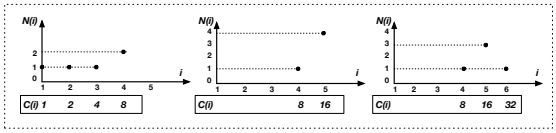
for fixed m and n, $\mathbb{C}_m$ is unique and so is $N_m(c_i)$. Hence, $S{=}S_m$ is unique for given m and n.

$$P^m_{Mult} = \sum_{i_{n-1}=1}^{m} \ldots \sum_{i_1=1}^{m} \sum_{i_0=1}^{m} 2^{i_1+i_1+\ldots+i_m} (a_0^{i_1} a_1^{i_2} \ldots a_{n-1}^{i_m}) \tag{6.4}$$

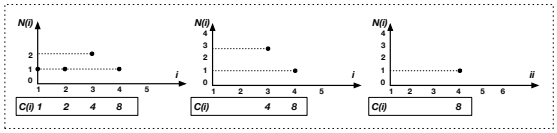
□

6.3.2 Example - Single Spectrum Function

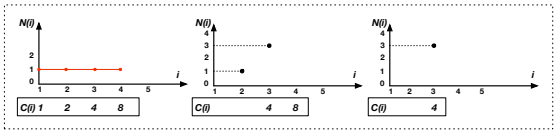
Using the spectral method, the equivalence checking problem can be solved by comparing k-spectrum of two different arithmetic designs, with $k = 1$ for linear arithmetic functions, $k \geq 2$ for non-linear arithmetic functions. If a reference design is not provided, the function will be abstracted by comparing the extracted spectrum to a set of known spectra. Parameter k is determined depending on the expected arithmetic function. For example, to solve a *combinational equivalence checking* or



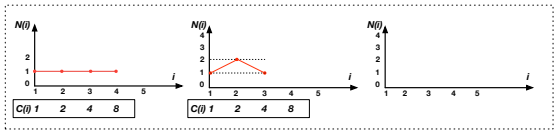
a First recorded spectrum.



b Second recorded spectrum.



c Third recorded spectrum.



d Final recorded spectrum.

Figure 6.3: Spectra of a 2-bit MAC of the four recorded expressions.

word-level abstraction problem of integer adders, k is equal to one. For multiplication with two (three) operands, k is equal to two (three).

To illustrate the uniqueness of the spectra, we record four intermediate expressions obtained by our *function extraction* tool [31] of a Booth multiplier and a CSA multiplier, shown in Figures 6.2(a) - (d). Initial spectrum of two multipliers are identical 1-*var* spectrum, since the polynomial expressions are both $z_0+2z_1+4z_2+8z_3+16z_4+32z_5$, corresponding to the word of the results. We can see that the intermediate spectra are different even thought the circuits are functionally equivalent. This is because the multiplication is implemented using a different algorithm and the intermediate expressions are different. However, the spectra of two designs at the primary inputs will match the spectrum of multiplication. For combinational equivalence checking (CEC) purpose, we can compare the spectra of two designs to check whether they are equivalent. For word-level abstraction, the spectrum at a cut that represents an arithmetic function is the point at which the abstraction process terminates. This means that, although the function of the circuits is unknown, the equivalence checking problem and the abstraction problem can be solved using this approach. Note that *function extraction* is only used for illustration purposes. The polynomial-time algorithm for extracting the spectrum is given in the next section.

6.3.3 Example - Multiple-Spectrum Function

An example of using multiple spectra to abstract the arithmetic function with multiple arithmetic operations is shown in Figure 6.3 for a 2-bit MAC ($F = A \times B + C$). The design was synthesized and mapped by the ABC system using command *strash; dch -v; map*. In this example, 1-*var*, 2-*var* and 3-*var* spectra are used for abstraction. The three algebraic spectra at each step represent the coefficient distribution of the 1-*var*, 2-*var*, and 3-*var* monomials, respectively. The initial expression is the output signature, i.e. $z_0 + 2z_1 + 4z_2 + 8z_3 + 16z_4$, which is a linear word.

To show how the multiplication and addition are identified, four intermediate expressions are recorded during the function extraction process. These are recorded during the abstraction process sequentially. As we can see, Figures 6.3(a) - (b) do not provide any evidence of identifying arithmetic operation, but Figure 6.3(c) and (d) do. A linear function is identified in the third expression in Figure 6.3(c). An additional non-linear arithmetic function is identified in Figure 6.3(d), by observing the spectrum of arithmetic multiplication. Since the entire function of the design is represented as the sum of k-var spectra, i.e., S_1+S_2, it indicates that its arithmetic function is adding one 4-bit number to the result of a 2-bit multiplication.

One important property of algebraic spectrum is *overridability*. Let the degree of m operands multiplication to be m, and the degree of addition to be 1. The spectrum of a given arithmetic function is always overridden by the highest degree arithmetic function. For example, given an arithmetic function F_1=A×(B+C), the spectrum S=S_2 since S_1 is empty. This means that the spectrum of arithmetic addition has been overridden by the highest degree F_1, which is 2. This can be seen by flattening the factor form $A \times (B + C)$ into a polynomial form F_2=$A \times B + A \times C$. As mentioned earlier, the sum of all k-*var* spectra represent the polynomial expression P. Since F_1=F_2, and $\underline{F_2 = S_2^{A \times B} + S_2^{A \times C}}$, the spectrum of F_1 S=S_2, which has been overridden by F_2. The spectra of F_1 and F_2, and the spectrum of a 2-operand 3-bit multiplication

F_0, are shown in Figure 6.4. We can see that the spectra of F_1 and F_2 is $2 \cdot S(F_0)$, i.e., $N_2(c_2^i) = N_2(c_2^i) = 2 \cdot N_2(c_0^i)$.

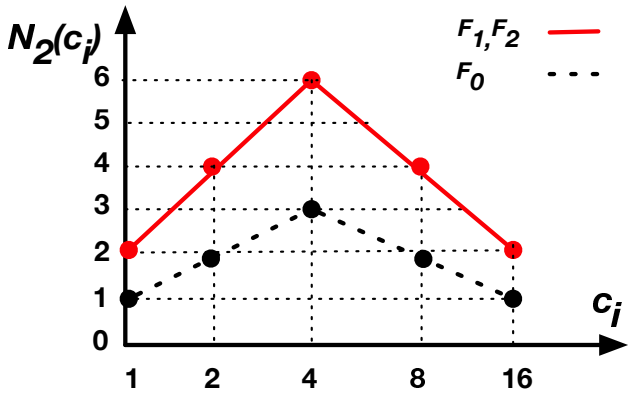


Figure 6.4: Spectra of F_1=$A \times (B + C)$, F_2=A×B+A×C, and F_3=A×B. A,B, and C are 3-bit unsigned words.

6.4 Polynomial-Time Spectrum Extraction

In the previous section we demonstrated that algebraic spectrum can be used to identify arithmetic functions, and that such a spectrum of arithmetic circuits can be obtained by *function extraction*. However, this approach is not efficient because its complexity is the same as the complexity of extracting the polynomial specification of the circuits. In this section, we introduce a new algorithm that extracts the algebraic spectrum of the gate-level arithmetic circuit from its AIG representation. The adder-tree constructed by HAs/FAs can be detected by identifying the pairs of XOR and MAJ functions from post-synthesized gate-level arithmetic circuits using AIG (see Chapter 5). In other words, any post-synthesized arithmetic circuit can be transformed into a netlist with HAs, FAs, and connecting logic gates. The weight of the *carry* bit of HA/FA is always double of the weights of the inputs, and the weight of the *sum* bit is the same as the inputs. Hence, the output weight of the linear blocks, such as HAs and FAs, can be propagated directly. Then, the algebraic

spectrum can be extracted without even performing algebraic rewriting from POs to PIs.

This approach is described in Algorithm 1. It takes the gate-level netlist of arithmetic function and the binary encoding of the output bits and produces the corresponding algebraic spectrum by propagating the weights using AIG. Using the approach described in Chapter 5, we first convert the gate-level netlist into AIG and detect all possible HAs and FAs. Then, the netlist only includes HAs, FAs, and ANDs in the AIG, sorted in reversed topological order (line 1-3). We initialize *k-var* spectra, k={1,2,3,...,n}, where n is determined based on the expected function degree of the circuit (line 4). For example, if the expected function is a two(three)-operand multiplication, then $n = 2(3)$. Then, we propagate the weights of all the signals from POs to PIs, through HAs, FAs and the AND gates. One important observation is that arithmetic multipliers are always implemented with an adder-tree and a partial product generator. This means that the weights of the signals can be propagated until they reach the partial product generator logic.

Algorithm 3 Spectrum Extraction with AIG

Input: Gate-level netlist; binary encoding of output bits
Output: Algebraic spectrum

1: Convert gate-level netlist to AIG *G(V, E)*.
2: Detect HAs/FAs in *G(V, E)*, sorted in topological order
3: $m \leftarrow$ number of HAs/FAs detected; $i = 1$
4: Initialize spectra at PO, denoted S_1^0, S_2^0, ..., S_n^0
5: **while** i $\leq$ m **do**
6: $\quad S_{1,2,...,n}^{i+1} \leftarrow S_{1,2,...,n}^{i}$ using i^{th} HA/FA; i++
7: **end while**
8: **return** S_1, S_2, ..., S_n

Example 3 (2-bit CSA-multiplier): We illustrate our algorithm using the example of a 2-bit multiplier shown in Figure 6.5. The extracted logic, with *gates (18, 16)*, and *gates (14, 12)*, forms two HAs where the outputs of 16 and 12 are the *carry* bits, and outputs of 18 and 14 are the *sum* bits. Assume that the weights w_i (encoding) of the output bits are known, i.e., w_i=2^i. Then, the weights of all

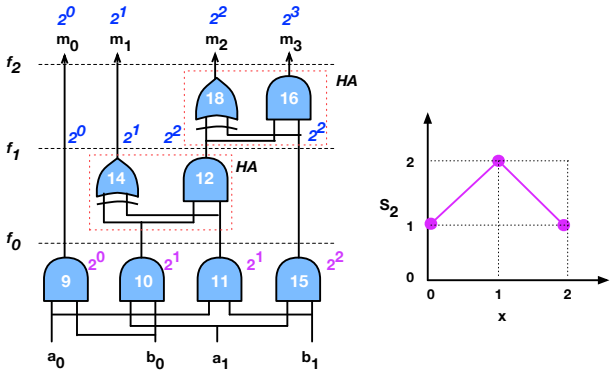


Figure 6.5: Extracting the function of 2-bit multiplier using spectral method without algebraic rewriting.

the signals in the netlist will be propagated from PO to PI in a reversed topological order. First, Algorithm 1 propagates the weights of the HA consisting of gates 18 and 16 to cut f_1 (Figure 6.5). The weight of gate 16 is 2^3, and weight of gate 18 is 2^2. This means that gate 18 produces the *sum* function. Since the input weights of an HA is the same as its *sum* bit, the two inputs of gates 18 and 16 must have weight 2^2. Similarly, at cut f_0, the weights of inputs of gates 14 and 12 are 2^1. Our algorithm terminates at this point since there are no more HA or FA nodes. While all the remaining logic gates are two-input ANDs, the 2-*var* spectrum is the only spectrum extracted (other spectra are empty). We can see that such an extracted spectrum matches the spectrum of a 2-bit multiplication.

Notice that the partial product generators differ depending on the multiplication algorithm used in constructing the multiplier circuits. For example, CSA-multiplier uses an AND-array, while Booth-multiplier requires implementing recoded partial products. But, regardless of the multiplier type, as soon as the adder-tree is detected, the algebraic spectrum can always be extracted using Algorithm 1. We illustrate our approach for Booth-encoded multiplier using a 3-bit radix-4 Booth-multiplier.

Example 4 (3-bit radix-4 Booth-multiplier): Because of the encodings of the partial products are encoded, the partial product generator logic includes more logic

functions than just AND function. Hence, the intermediate polynomial expressions of the partial products may be presented with more than one monomial. For example, the polynomial expressions of all the partial products of 3-bit radix-4 Booth multiplier are shown in Equation 6.5. After applying Algorithm 1, we can get the weights of all the partial products, but the spectrum cannot be extracted directly. However, we observe that, the monomials that are not 2-*var* monomials can be considered as redundant. This is because they can be canceled according to their weights. For example, $a_2b_1b_2$ exists in pp_{31} and pp_{21}. Since the function of the design is the sum of all the partial products with correct weights, $2^4a_2b_1b_2+2^3(-2a_2b_1b_2)=0$. Note that this is not the assumption of our approach. This information is obtained since the adder-tree has been detected. Hence, the algebraic spectrum can be extracted by just collecting all the 2-var monomials, which matches the multiplication spectrum.

$$
\begin{aligned}
2^4 &: pp_{31} = \underline{a_2b_1b_2} \\
2^3 &: pp_{21} = \underline{-2a_2b_1b_2} + \underline{a_1b_1b_2} + a_2b_1 + a_2b_2 \\
2^2 &: pp_{11} = \underline{-2a_1b_1b_2} + \underline{a_0b_1b_2} + a_1b_1 + a_1b_2 \\
2^1 &: pp_{01} = \underline{-2a_0b_1b_2} + a_0b_1 + a_0b_2 \\
2^3 &: pp_{30} = \underline{a_2b_0b_1} - a_2b_1 \\
2^2 &: pp_{20} = \underline{-2a_2b_0b_1} + \underline{a_1b_0b_1} - a_1b_1 + a_2b_0 \\
2^1 &: pp_{10} = \underline{-2a_1b_0b_1} + \underline{a_0b_0b_1} - a_0b_1 + a_1b_0 \\
2^0 &: pp_{00} = \underline{-2a_0b_0b_1} + a_0b_0
\end{aligned}
\tag{6.5}
$$

Table 6.1: Results of extracting the specification of pre- and post-synthesized CSA multipliers compared to *functional extraction* presented in [31]. **t(s)* is the runtime in seconds. **mem* is the memory usage in mb.

#bits	*Pre-synthesized*				*Post-synthesized*			
	[31]		This approach		[31]		This approach	
	t(s)	mem	t(s)	mem	t(s)	mem	t(s)	mem
64	1.89	74	0.04	34	33.50*	MO	0.08	34
128	8.12	288	0.15	117	-	MO	0.46	120
256	32.65	1157	0.82	441	-	MO	6.96	439
512	130.22	4427	3.76	1695	-	MO	28.70	1876

6.5 Results

The technique described in this paper has been implemented in C++ integrated with ABC [80]. It takes the gate-level netlist (Verilog or BLIF files) and produces an algebraic spectrum. The experiments include extracting the arithmetic function from the gate-level netlist, and abstracting the word-level operations. The experiments are obtained using a set of arithmetic circuits synthesized by ABC. We use two multiplication algorithms for implementing those circuits, namely Carry-Save-Adder (CSA) multiplier and Radix-4 Booth multiplier. The verification results are compared with the state-of-the-art rewriting-based approach [31]. We will add the comparison with the Grobner Basis based approach [100] if we get the tool from the authors. For word-level abstraction, we compare our approach with simulation graph based approach [105] and our function extraction approach [129]. The comparison with the contemporary formal methods such as SAT, SMT, ABC(cec), and the commercial tools, are not directly provided in this paper. The computer algebraic methods have been demonstrated to be magnitude orders faster than those techniques [31][100].

Table 6.2: Results of extracting the specification of the post-synthesized complex arithmetic circuits compared to *functional extraction* presented in [31]. *MO = Memory out of 8 GB.

Benchmarks (256-bit)	[31]		This approach	
	runtime(s)	mem(MB)	runtime(s)	mem(MB)
$F=A\times B+C$	-	MO	10.3	447
$F=A\times(B+C)$	-	MO	11.1	451
$F=A\times B\times C$	-	MO	67.5	2871
$F=1+A+A^2+A^3$	-	MO	77.1	3331

Table 6.3: Runtime of extracting the specification of the radix-4 Booth multiplier. *MO = Memory out of 8GB.

# bits	[31]		This approach	
	Pre-syn (s)	Post-syn (s)	Pre-syn (s)	Post-syn (s)
64	MO	MO	0.05	0.06
128	-	-	0.51	6.97
256	-	-	1.95	22.70
512	-	-	5.50	-

Table 6.4: Evaluation of word-level abstraction using algebraic spectrum. Multiplications in F_1 and F_2 are implemented using CSA-multiplier. F_3 uses radix-4 Booth-multiplier.

Benchmarks 128-bit	This Approach			[129]			[105]		
Function (128-bit)	*Mult*	*Add*	Runtime	*Mult*	*Add*	Runtime	*Mult*	*Add*	Runtime
F_1=$A\times B+C$ *(CSA)*	1	1	0.45 s	1	1	23760 s	0	0	-
F_2=$A\times B+A\times C$ *(CSA)*	2	1	1.03 s	2	1	48560 s	0	0	-
F_3=$A\times B+C$ *(Booth)*	-	-	-	-	-	-	0	0	-

The experiments were conducted on a PC with Intel(R) Xeon CPU E5-2420 v2 2.20 GHz x12 with 32 GB memory.

Evaluations of verifying pre-synthesized and post-synthesized CSA multipliers are shown in Tables 6.1, 6.2 and 6.3. The same results for post-synthesized complex unsigned arithmetic circuits taken from [31] are shown in Table 6.2. Note that the circuits in Table 6.1 are optimized using *(resyn, resyn2; dch; map)* with complex standard cell library. The runtime and memory usage are compared to *functional extraction* [31]. Synthesis was shown to have significant impact on function extraction approach [125]. That approach can only extract the specification of the pre-synthesized and slightly synthesized (with reduced simple technology library), but cannot be applied to heavily optimized circuits. The reason for that is the lack of sufficient in heavily optimized circuits, which caused memory explosion. This is shown in Tables 6.1 and 6.2. In Table 6.2, the functions of the arithmetic circuits are shown in the first column. The bit-width varies between 64 and 512 bits. We can see that the runtime of the proposed approach is less than 30 seconds for both the pre- and post-synthesized CSA multipliers of any bit-width. For example, extracting post-synthesized 64-bit multiplier using functional extraction reaches the 8 GB memory limit (MO=8GB) in 33 seconds. However, extracting the specification using the algebraic spectrum of the 512-bit post-synthesized multipliers requires only 1.8 GB memory and produces the result within 30 seconds. Additionally, our ap-

proach outperforms *functional extraction* on complex arithmetic circuits (Table 6.2) and Booth-multipliers (Table 6.3).

Results of abstracting word-level operations from gate-level arithmetic circuits are shown in Table 6.4. We use three types of circuits that are constructed with multiplication and addition. The multiplications are implemented using CSA-multiplier (F_1 and F_2) and Booth-multiplier (F_3). We can see that our approach can identify the word-level operations in just one second for F_1 and F_2. In contrast, the approach of [105] cannot tell whether there exists multiplication or addition in the circuits. Currently, our approach can extract the entire spectrum of F_3, but it cannot identify if there is one multiplier and one adder in the circuit of F_3. The reason for this is that the detected adders of the Booth-multiplier are not clearly separated with the detected adders of the adder.

6.6 Conclusion

This chapter presented a novel computer algebraic approach to abstract the word-level information from the gate-level netlist. In contrast to [66][113][105], it can address the problem of abstracting words from a large non-linear gate-level arithmetic circuit (MAC). This approach is based on a new canonical representation of arithmetic functions, called *algebraic spectrum.* that uses coefficient distribution of a polynomial expression to describes the arithmetic function. Such defined algebraic spectrum has been proved to be canonical if the arithmetic function is constructed with any combination of addition and multiplication in Section 6.4.1.

CHAPTER 7

FORMAL ANALYSIS OF FINITE FIELD ARITHMETIC CIRCUITS

7.1 Introduction

Galois field (GF) is a number system with a finite number of elements and two main arithmetic operations, addition and multiplication; other operations can be derived from those two [85]. It has been extensively applied in many digital signal processing and security applications, such as Elliptic Curve Cryptography (ECC), Advanced Encryption Standard (AES). For example, the S-BOX and MixColumn transforms are treated as field GF(2^8) and the internal data is performed by arithmetic operations in GF(2^8), including addition and multiplication in field. Finite field multiplication is the most complex operation in circuit design and verification, and is widely used in many applications. Specifically, in cryptography systems, the size of Galois field circuits can be very large. Therefore, developing general formal analysis techniques of Galois field arithmetic HW/SW implementations becomes critical.

The elements in field GF(2^m) can be represented using polynomial rings. The field of size m is constructed using *irreducible polynomial* $P(x)$, which includes terms of degree d with coefficients in GF(2), with $d \in$ [0, m]. For example, $P(x)$=x^4+x+1 is an irreducible polynomial in GF(2^4). The multiplication in the field is performed modulo $P(x)$. Theoretically, there is a large number of irreducible polynomials available for constructing the field arithmetic operations in GF(2^m). However, the irreducible polynomial has great impact on the actual implementation of the resulting GF circuits and the performance of field arithmetic operations. It differs in the number of bit-level XOR operations. It is believed that, in general, the irreducible polynomial with

minimum number of elements gives the best performance [33]. However, recently some work [102] demonstrate that the best irreducible polynomial from circuit performance point of view varies depending on computer architecture in which it is used, such as ARM vs. Intel-Pentium. In other words, 1) for GF(2^m) multiplication, each irreducible polynomial corresponds to a unique implementation; 2) for a fixed field size, there exist many irreducible polynomials that could be used for constructing the field in different applications.

Due to the rising number of threats in hardware security, analyzing finite field circuits becomes very important. Computer algebra techniques with polynomial representations is believed to offer best solution for analyzing arithmetic circuits [70][89][100][31]. These works address the verification problems and abstraction problems of Galois field arithmetic and integer arithmetic implementations [89][100][31]. Specifically, symbolic computer algebra methods have also been used to reverse engineer the word-level operations for GF circuits and integer arithmetic circuits to speed up the verification performance [129][101][91]. In Chapter 6, we proposed an original spectral method based on analyzing the internal algebraic expressions during the rewriting procedure. A. Sayed-Ahmed et. al [101] introduced a reverse engineering technique in Algebraic Combinational Equivalence Checking (ACEC) process using *Gröbner Basis* by converting the function into canonical polynomials. However, both techniques are applicable to integer arithmetic only. An abstraction technique over GF(2^m) is introduced by analyzing the polynomial representation[91].

However, there are two limitations of the above mentioned algebraic techniques:

- **1)** They are all restricted to the implementation with known binary encoding of the inputs and output.
- **2)** None of those algebraic techniques can be applied in parallel. Additionally, the abstraction and verification techniques for GF(2^m) arithmetic circuits, requires a known irreducible polynomial $P(x)$ [70][91].

In this chapter, we present a formal approach that can reverse engineer the gate-level finite field arithmetic circuits and extract the polynomial signature of the circuits simultaneously. Specifically, this chapter first introduces a parallel algebraic rewriting approach for extracting the polynomial signature of finite field arithmetic circuit (Section 7.3). Second, using the parallel verification framework, we present a reverse engineering approach (Section 7.4) that extracts the polynomial specification of a gate-level GF(2^m) multiplier, when the bit positions of input and output bits and the irreducible polynomial used for constructing the multiplication are unknown. The verification and reverse engineering approaches are demonstrated using bit-blasted GF(2^m) *Mastrovito* and *Montgomery* multipliers up to 571-bit width, implemented using various irreducible polynomials.

7.2 Background

7.2.1 Galois Field Multiplication

Galois field (GF) is a number system with finite number of elements and two main arithmetic operations, addition and multiplication; other operations such as division can be derived from those two [85]. Galois field with p elements is denoted as GF(p). Prime field, denoted GF(p), is a finite field consisting of finite number of integers $\{1, 2, ..., p-1\}$, where p is a prime number, with additions and multiplication performed *modulo p*. Binary extension field, denoted GF(2^m) (or $\mathbb{F}_{2^m}$), is a finite field with 2^m elements. Unlike in prime fields, however, the operations in extension fields are not computed *modulo* 2^m. Instead, in one possible representation (called *polynomial basis*). Multiplication of field elements is performed modulo *irreducible polynomial* $P(x)$ of degree m and coefficients in GF(2). The irreducible polynomial $P(x)$ is analog to the prime number p in prime fields $GF(p)$. Extension fields are used in many cryptography applications, such as AES and ECC. In this work, we focus on the verification problem of GF(2^m) multipliers.

Two different GF multiplication structures, constructed using different irreducible polynomials $P_1(x)$ and $P_2(x)$, are shown in Figure 7.1. The integer multiplication takes two n-bit operands as input and generates a $2n$-bit word, where the values computed at lower significant bits ripple through the carry chain all the way to the most significant bit (MSB). In contrast, in GF(2^m) implementations there is no carry chain, and the number of outputs is reduced to n using irreducible polynomial P(x). For example, to represent the result in GF(2^4), with only four output bits, the four most significant bits in the result of the integer multiplication have to be reduced to GF(2^4). The result of such a reduction is shown in Figure 7.1. In GF(2^4), the input and output operands are represented using polynomials $A(x)$, $B(x)$ and $Z(x)$, where $A(x)=\sum_{n=0}^{n=3} a_n x^n$, $B(x)=\sum_{n=0}^{n=3} b_n x^n$, and $Z(x)=\sum_{n=0}^{n=3} z_n x^n$, respectively.

Example 1: The functions of s_i ($i \in [0, 6]$) are represented using polynomials in GF(2), namely: $s_0=a_0b_0$, $s_1=a_1b_0+a_0b_1$, ..., up to $s_6=a_3b_3$[1]. The outputs z_n (n $\in$ [0, 3]) are computed modulo the irreducible polynomial $P(x)$. Using $P_2(x)=x^4+x+1$, we obtain : $z_0=s_0+s_4$, $z_1=s_1+s_4+s_5$, $z_2=a_0b_2+a_1b_1+a_2b_0+a_2b_3+a_3b_2+a_3b_3$, and $z_3=a_0b_3+a_1b_2+a_2b_1+a_3b_0+a_3b_3$. The coefficients of the multiplication results are shown in Figure 7.2. In digital circuits, partial products are implemented using AND gates, and addition modulo 2 is done using XOR gates. Note that, unlike in integer multiplication, in GF(2^m) circuits there is no carry out to the next bit. For this reason, as we can see in Figure 7.1, the function of each output bit can be computed independently of other bits.

7.2.2 Irreducible Polynomials

For constructing the field GF(2^m), the irreducible polynomial can be either a trinomial, x^m+x^a+1, or a pentanomial $x^m+x^a+x^b+x^c+1$ [83]. Typically, the pentanomial is chosen as irreducible polynomial only if an irreducible trinomial doesn't

[1]For polynomials in GF(2), "+" is computed as modulo 2.

			a_3	a_2	a_1	a_0
			b_3	b_2	b_1	b_0
			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			
s_6	s_5	s_4	s_3	s_2	s_1	s_0

$P(x)_1$=$x^4 + x^3 + 1$

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

$P(x)_2$=$x^4 + x + 1$

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

Figure 7.1: Two multiplications in GF(2^4) constructed using $P(x)_1$=$x^4 + x^3 + 1$ and $P(x)_2$=$x^4 + x + 1$.

output	polynomial expression
z_0	$(a_0b_0)+a_1b_3+a_2b_2+a_3b_1$
z_1	$(a_0b_1+a_1b_0)+a_1b_3+a_2b_2+a_2b_3+a_3b_1+a_3b_2$
z_2	$(a_0b_2+a_1b_1+a_2b_0)+a_2b_3+a_3b_2+a_3b_3$
z_3	$(a_0b_3+a_1b_2+a_2b_1+a_3b_0)+a_3b_3$

Figure 7.2: Extracted algebraic expressions of the four output bits of a GF(2^4) multiplier. P(x)=x^4+x+1.

exist [83]. In order to obtain efficient GF multiplication algorithm, it is required that m - $a \geq w$. However, as demonstrated in [102], the trinomials are not always better than pentanomials. It means that for a given field size, there are various irreducible polynomials that can be used.

An example of constructing GF(2^4) multiplication using two different irreducible polynomials is shown in Figure 7.1. We can see that each polynomial corresponds to a unique multiplication result. The performance difference can be evaluated by counting the number of XOR operations in each multiplication. Since the number of AND and XOR operations for generating partial products (variables s_i in Figure 7.1) is the same, the difference is only caused by the reduction of the corresponding

polynomials modulo $P(x)$. The number of XOR operations introduced by reduction process can be counted as the number of terms in each column minus one. For example, the number of XORs using $P_1(x)$ is 3+1+2+3=9; and using $P_2(x)$, the number of XORs is 1+2+2+1=6.

As will be shown in the next section, given the structure of the GF(2^m) multiplication, such as the one shown in Figure 7.1, one can readily identify the irreducible polynomial $P(x)$. This can be done by extracting the terms s^k corresponding to the entry s^m (here s^4) in the table and generating the irreducible polynomial beyond x^m. We know that $P(x)$ must contain x^m, and the remaining terms x^k of $P(x)$ are obtained from the non-zero terms corresponding to the entry s^m. For example, for the irreducible polynomial $P_1(x) = x^4 + x^3 + x^0$, the terms x^3 and x^0 are obtained by noticing the placement of s^4 in columns z_3 and z_0. Similarly, for $P_2(x) = x^4 + x^1 + x^0$, the terms x^1 and x^0 are obtained by noticing that s^4 is placed in columns z_1 and z_0. The reason for it and the details of this procedure will be explained in the next section.

7.3 Parallel Extraction in Galois Field

In this section, we introduce our method for extracting the unique algebraic expressions of the output bits (e.g. Figure 7.2) using computer algebraic method. This can be used to verify the GF(2^m) multipliers when the binary encoding of inputs and output and the irreducible polynomial are given. We introduce a parallel function extraction framework in GF(2^m), which allows us to individually extract the algebraic expression of each output bit. This framework is used for reverse engineering, since our reverse engineering approach is based on analyzing the algebraic expression of output bits in GF(2), as introduced in Section I.

7.3.1 Computer Algebraic model

The circuit is modeled as a network of logic elements of arbitrary complexity including: basic logic gates (AND, OR, XOR, INV) and complex standard cell gates (AOI, OAI, etc.) obtained by synthesis and technology mapping. Instead of modeling Boolean operators using pseudo-Boolean equations, we use the algebraic models in $GF(2)$, i.e. modulo 2. For example, the pseudo-Boolean model of XOR(a, b)=$a + b$ $-2ab$ is reduced to $(a + b + 2ab)$ mod 2 = $(a + b)$ mod 2. The following algebraic equations are used to describe basic logic gates in $GF(2^m)$, according to [70]:

$$\begin{aligned} \neg a &= 1 + a \\ a \wedge b &= a \cdot b \\ a \vee b &= a + b + a \cdot b \\ a \oplus b &= a + b \end{aligned} \tag{7.1}$$

7.3.2 Outline of the Approach

Similarly to function extraction presented in Chapter 3, the computed function of the circuits is specified by two polynomials. The *output signature* of a $GF(2^m)$ multiplier, $Sig_{out} = \sum_{i=0}^{m-1} z_i x^i$, with $z_i \in GF(2)$. The *input signature* of a $GF(2^m)$ multiplier, $Sig_{in} = \sum_{i=0}^{m-1} \mathbb{P}_i x^i$, with coefficients $\mathbb{P}_i \in GF(2)$ being product terms, and addition operation performed modulo 2 (e.g. $(a_0 b_0 + a_1 b_1)$ mod 2). Note that $\mathbb{P}_i$ is a partial product set, which is a series of XOR operations with partial products p_i as inputs, e.g., in GF(2^4), $\mathbb{P}_1$=$p_{10} \oplus p_{01}$, p_{10}=$a_1 b_0$, p_{01}=$a_0 b_1$. For a $GF(2^m)$ multiplier, if the irreducible polynomial $P(x)$ is provided, Sig_{in} is known. Our goal is to transform the output signature, Sig_{out}, using polynomial representation of the internal logic elements, into the input signature Sig_{in} in $GF(2^m)$. The the goal of the verification problem is then to check if $Sig_{in} = Sig_{out}$, expressed in the primary inputs.

Theorem 1: *Given a combinational arithmetic circuit in $GF(2^m)$, composed of logic gates, described by Eq. 1, input signature Sig_{in} computed by backward rewriting is unique and correctly represents the function implemented by the circuit in $GF(2^m)$.*

Proof: The proof of correctness relies on the fact that each transformation step (rewriting iteration) is correct. That is, each internal signal is represented by an algebraic expression, which always evaluates to a *correct value* in $GF(2^m)$. This is guaranteed by the correctness of the algebraic model in Eq. (7.1), which can be proved easily by inspection. For example, the algebraic expression of *XOR(a,b)* in $\mathbb{Z}_{2^m}$ is $a + b - 2ab$. When implemented in $GF(2^m)$, the coefficients in the expression must be in $GF(2)$. Hence, *XOR(a,b)* in $GF2^m$ is represented by $a + b$. The proof of uniqueness is done by induction on i, the step of transforming polynomial F_i into F_{i+1}. A detailed induction proof for expressions in $\mathbb{Z}_{2^m}$ is provided in Chapter 3.

□

Algorithm 4 Backward Rewriting in $GF(2^m)$

Input: Gate-level netlist of $GF(2^m)$ multiplier
Input: Output signature Sig_{out}, and (optionally) input signature, Sig_{in}
Output: GF function of the design, and answer whether Sig_{out}==Sig_{in}

```
1: P={p_0, p_1, ..., p_n}: polynomials representing gate-level netlist
2: F_0=Sig_out
3: for each polynomial p_i ∈ P do
4:     for output variable v of p_i in F_i do
5:         replace every variable v in F_i by the expression of p_i
6:         F_i → F_{i+1}
7:         for each element/monomial M in F_{i+1} do
8:             if the coefficient of M%2==0
9:             or M is constant, M%2==0 then
10:                remove M from F_{i+1}
11:            end if
12:        end for
13:    end for
14: end for
15: return F_n and F_n =?Sig_in
```

Theorems 1, together with the algebraic model in Eq. (1), provide the basis for polynomial reduction in backward rewriting in this work. This is described by

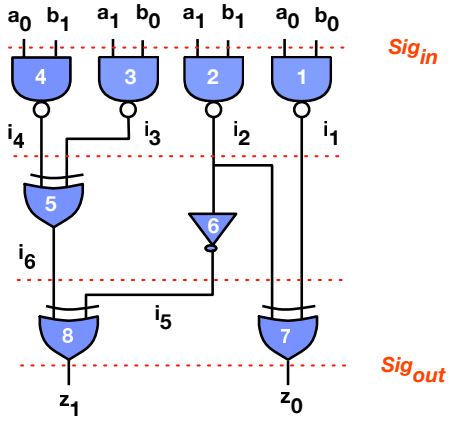


Figure 7.3: The gate-level netlist of post-synthesized and mapped 2-bit multiplier over $GF(2^2)$. The irreducible polynomial $P(x) = x^2 + x + 1$.

Algorithm 4. Our method takes the gate-level netlist of a $GF(2^m)$ multiplier as input and first converts each logic gate into equations using Eq. (1). The output signature Sig_{out} is required to initialize the backward rewriting. The rewriting process starts with $F_0 = Sig_{out}$, and ends when all the variables in F_i are primary inputs. This is done by rewriting the polynomials representing logic elements in the netlist in reversed topological order as discussed in Chapter 3. Each iteration includes two steps:

- **Step 1:** substitute the variable of the gate output using the expression in the inputs of the gate (Eq.1), and name the new expression F_{i+1} (lines 3 - 6).
- **Step 2:** simplify the new expression (modulo 2) by removing all the monomials (including constants) that evaluate to 0 in $GF(2)$ (line 3 and lines 7 - 10).

The algorithm outputs the function of the design in $GF(2^m)$ after n iterations, where n is the number of gates in the netlist. The final expression F_n can be used for functional verification, by checking if it matches the expected input signature (if provided).

Example 2 (Figure 7.3): We illustrate our method using a post-synthesized 2-bit multiplier in $GF(2^2)$, shown in Figure 7.3. The irreducible polynomial is $P(x)$

Sig_{out}: F_{init}=z_0+xz_1	Eliminating terms
G8: F_8=z_0+$x(i_5+i_6)$	-
G7: F_7=i_1+i_2+$x(i_5+i_6)$	-
G6: F_6=i_1+i_2+$x(i_3+i_4+i_5)$	-
G5: F_5=i_1+i_2+$x(i_3+i_4+i_2+1)$	-
G4: F_4=i_1+i_2+$x(i_2+i_3+a_0b_1)$+2x	$2x$
G3: F_3=i_1+i_2+$x(i_2+a_1b_0+a_0b_1+1)$	-
G2: F_2=i_1+a_1b_1+1+$x(a_1b_1+a_1b_0+a_0b_1)$+2x	$2x$
G1: F_1=a_0b_0+a_1b_1+2+$x(a_1b_1+a_1b_0+a_0b_1)$	2
Sig_{in}: $a_0b_0+a_1b_1+x(a_1b_1+a_1b_0+a_0b_1)$	-

Figure 7.4: Function extraction of a 2-bit GF multiplier shown in Figure 7.3 using backward rewiring from PO to PI.

$= x^2 + x + 1$. The output signature is $Sig_{out} = z_0+z_1x$, and input signature is $Sig_{in} = (a_0b_0+a_1b_1)+(a_1b_1+a_1b_0+a_0b_1)x$. First, $F_{init} = Sig_{out}$ is transformed into F_8 using polynomial of gate $g8$, z_1=i_5+i_6 and simplified to $F_8 = z_0+i_5x+i_6x$. Then, the polynomials F_i are successively derived from F_{i+1} and checked for a possible reduction. The first reduction happens when F_5 is transformed into F_4, where i_4 (at gate g_4) is replaced by $(1 + a_0b_0)$. After simplification, a monomial $2x$ is identified and removed by modulo 2 from F_4. Similar reductions are applied during the transformations $F_3 \rightarrow F_2$ and $F_2 \rightarrow F_1$. Finally, the function of the design is extracted as expression F_1. A complete rewriting process is shown in Figure 7.4. We can see that $F_1 = Sig_{in}$, which indicates that the circuit indeed implements the $GF(2^2)$ multiplication with $P(x)$=$x^2 + x + 1$.

An important observation is that the polynomial reductions happen in a logic cone of every output bit *independently* of other bits, regardless of logic sharing between the cones. For example, the reductions in F_4 and F_2 happen within the logic cone of output z_1 only. Similarly, in F_1, the reduction is within logic cone of z_0.

Theorem 3: *Given a $GF(2^m)$ multiplier with $Sig_{out} = F_0 = z_0x^0 + z_1x^1 + ... + z_mx^m$; let F_i=$E_0x^0 + E_1x^1 + ... + E_mx^m$, where E_i is an algebraic expression*

in $GF(2)$ obtained during rewriting. Then, the polynomial reduction is possible only within a single expression E_i, for i=1, 2, ..., m.

Proof: Consider a polynomial $E_i x^{i_i} + E_k x^{i_k}$, where E_i and E_k are simplified in $GF(2)$. That is, $E_i = (e_i^1 + e_i^2 + ...)$, and $E_k = (e_k^1 + e_k^2 + ...)$. After simplifying each of the two polynomials, there are no common monomials between $E_i x^{i_i}$ and $E_k x^{i_k}$. This is because for any element, $e_i^l x^{i_i} \neq e_k^j x^{i_k}$, for any pairs of (i, k) and (l, j). That is, the sum of such elements cannot be reduced by modulo 2.

□

7.3.3 Implementation

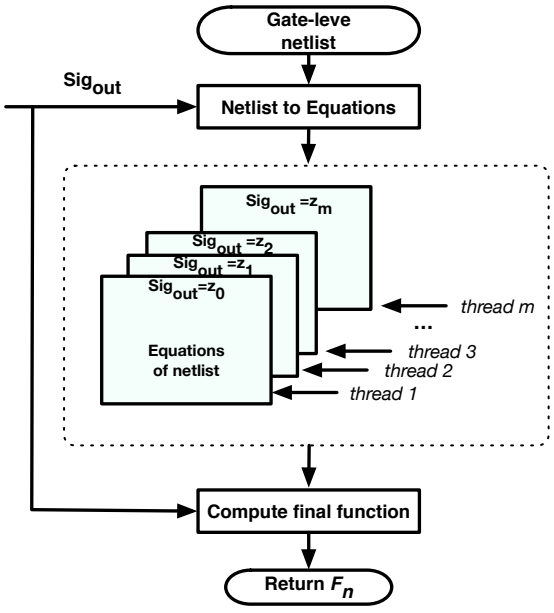


Figure 7.5: Overview of the parallel extraction framework.

This section describes the implementation of our parallel verification method for Galois field multipliers. Our approach takes the gate-level netlist as input, and outputs the extracted function of the design. It includes four steps:

1. **Netlist to equations:** parse the gate-level netlist into algebraic equations based on Equation 1. The equations are listed in reversed topological order, to

be rewritten by backward rewriting in the next step. m copies of this equation file will be made for a GF(2^m) multiplier.

2. **Generate signatures:** split the output signature of GF(2^m) multipliers into m polynomials with Sig_{out_i}=z_i. Insert the new *signatures* into the m copies of the equation file generated from Step1. Each signature is a single output bit.

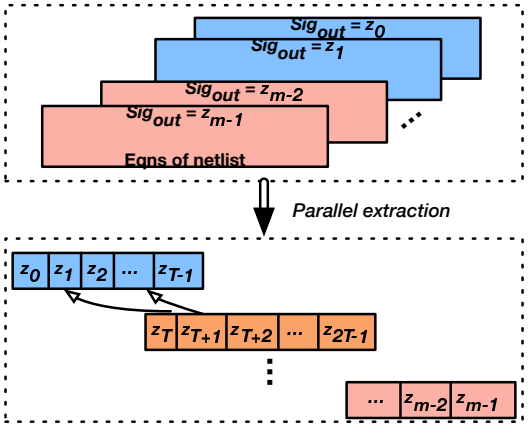


Figure 7.6: Step3: parallel extraction of a GF(2^m) multiplier with number of threads T.

3. **Parallel extraction:** apply Algorithm 4 on each equation file to extract the polynomial expression of each output in parallel. In contrast to the polynomial reductions shown in Chapters 3 and 5, the internal expression of each output bit does not offer any polynomial reduction (*monomial cancellations*) with other bits.

   Ideally, our approach can extract GF(2^m) multiplier in m threads. However, due to the limited computing resources, it is impossible to extract GF(2^m) multipliers for a large number of threads. Hence, our approach puts a limit on a number of parallel threads T. In particular T = 5, 10, 20 and 30 have been tested in this work. This process is described in Figure 7.6. The m extraction tasks are assigned into several task sets, ordered from LSB to MSB. In each set,

the extractions are processed in parallel. Since the runtime of each extraction within the set could be different, the tasks in the next set will start as soon as any previous task terminated.

4. **Step4: Finalization:** compute the final function of the multiplier. Once the algebraic expression of each output bit in $GF(2)$ is computed, our method computes the final function by constructing the Sig_{out} using the rewriting process in step 3.

Data Structure: Our algorithm uses an efficient data structure to support these simplifications and efficiently implement an iterative substitution and elimination process. Specifically, a data structure is maintained that records the terms (monomials) in the expression that contain the variable to be substituted. It reduces the cost of finding what terms will have their coefficients changed during the substitution. Each element represents one monomial consisting of the variables in the monomials, and its coefficient. The expression data structure is a C++ object that represents a pseudo-Boolean expression, which is the sum of all the elements in the data structure. It supports both fast addition and fast substitution with two C++ maps, implemented as binary search trees, a terms map and a substitution map. This data structure includes two cases of simplifications: 1) after substitution, the coefficients of all the monomials will be updated. The monomials with a 0 coefficient are eliminated; 2) according to Remark 2, the monomials which their coefficient modulo 2 evaluate to 0 are eliminated. Note that the second case is applied after each substitution.

Example 3 (Figure 7.7): We illustrate our parallel extraction method using the 2-bit multiplier in $GF(2^2)$ in Figure 7.3. The output signature $Sig_{out} = z_0+xz_1$ is split into two signatures, $Sig_{out0} = z_0$ and $Sig_{out1} = z_1$. Then, the rewriting process is applied to Sig_{out0} and Sig_{out1} in parallel. When Sig_{out0} and Sig_{out1} have been successfully extracted, the two signatures are merged as $Sig_{out0}+xSig_{out1}$ resulting in the polynomial Sig_{in}. In Figure 7.4, we can see that elimination happens three times

Sig_{out0}=z_0	elim	Sig_{out1}=x·z_1	elim
G8: z_0	-	G8: i_5x+i_6x	-
G7: i_1+i_2	-	G7: i_5x+i_6x	-
G6: i_1+i_2	-	G6: i_2x+x+i_6x	-
G5: i_1+i_2	-	G5: i_2x+x+i_3x+i_4x	-
G4: i_1+i_2	-	G4: i_2x+**x**+$i_3x+a_0b_1x$+**x**	2x
G3: i_1+i_2	-	G3: $i_2x+a_1b_0x$+x+a_0b_1x	-
G2: $i_1+a_1b_1+1$	-	G2: a_1b_1x+**x**+a_1b_0x+**x**+a_0b_1x	2x
G1: **1**+$a_0b_0+a_1b_1$+**1**	*2*	G1: x($a_1b_1+a_1b_0+a_0b_1$)	-
z_0=$a_0b_0+a_1b_1$, z_1=x($a_1b_1+a_1b_0+a_0b_1$)			

Figure 7.7: Extracting the algebraic expression of z_0 and z_1 separately in Figure 7.4.

(F_5, F_7, and F_8). According to Theorem 2, we know that the elimination happens within each element in GF(2^n). In Figure 7.7, one elimination in Sig_{out0} and two eliminations in Sig_{out1} have been done independently, as shown earlier (Example 2).

7.4 Reverse Engineering in Galois Field

In this section, we present our approach to reverse engineer any of GF(2^m) multipliers. Using the *extraction* technique presented in the previous section, we can extract the algebraic expression of each output bit. In contrast to the algebraic technique of presented for integer arithmetic circuits (Chapters 3 and 5), the extraction technique in GF(2^m) can extract the algebraic expression of each output bit independently. This means that the extraction can be done without the knowledge of the bit position of the inputs and output. We introduce two theorems to support this claim.

In a GF(2^m) multiplication, let s_i ($i \in \{0,1,...,2m\text{-}1\}$) be a set of products generated by the AND and XOR operations. For example, in Figure 7.1, there are six product sets, s_0, s_1, ..., s_6, where s_1=a_1b_0+a_0b_1; or written as a set : s_1=$\{a_1b_0, a_0b_1\}$, etc. The product sets *out of field* GF(2^m) (sets s_i whose index i=[m, 2m-1]) will be reduced into the field GF(2^m). This is the case for sets s_4, s_5, s_6 in Figure 7.1. We call these product sets ***out-field product sets***, and call the product sets s_i with i=[0,$m-1$] ***in-field product sets***. For example, in Figure 7.1, s_0, s_1, s_2, s_3 are *in-field product sets*, and s_4, s_5, s_6 are *out-field product sets*. The partial products

generation is the same as in the integer multiplication. For a GF(2^m) multiplication, m product sets are *in-field*, and $m-1$ product sets are *out-field*.

7.4.1 Output encoding determination

Theorem 3: Given a GF(2^m) multiplication, the in-field product sets (s_0, s_1, ..., s_{m-1}) are appearing only in one element in GF(2^m), and the out-field product sets (s_m, s_{m+1}, ..., s_{2m-1}) are appearing in at least two elements in GF(2^m), in the reduction process mod P(x).

Proof: In a Galois field multiplication, the in-field product sets (s_0, s_1, ..., s_{m-1}) are not required to be reduced since they are already in GF(2^m). Hence, the in-field product sets are generated only once in each output element. Regarding the out-field product sets, we prove that they are appearing in at least two elements using contrapositive proof. Let the statement $\mathbb{S}$ read: *the out-field product sets are only appearing once in reduction process mod $P(x)$*. Since each product in the product sets s_k is a k^{th} element, each product represents x^k in the polynomial ring of GF(2^m) multiplication. If $\mathbb{S}$ is *true*, then statement $\mathbb{S}'$ must be *true*.

$\mathbb{S}$: *The out-field product sets are only appearing in one element in mod P(x) reduction.*

$\mathbb{S}'$: *for $k \geq m$, x^k mod P(x) = x^{k-m}.*

We can see that statement $\mathbb{S}'$ is true if and only if $P(x)= x^m$. However, according to the definition of irreducible polynomial, $P(x)= x^m$ is an illegal irreducible polynomial. Hence, if statement $\mathbb{S}'$ is *false*, statement $\mathbb{S}$ is also *false*. Hence, $\neg\mathbb{S}$: *the out-field product sets are appearing in at least two elements in mod P(x) reduction.*, is *true*. Hence, theorem 4 is proved. □

Based on Theorem 3, we can find the in-field product sets, s_0, s_1, ..., s_{m-1}, by searching the unique products in the resulting algebraic expressions of the output bits. In this context, *unique products* are the products that exist in only one of

the extracted algebraic expressions. Since the in-field products set indicates the bit position of the output, we can determine the bit positions of the output bits as soon as all the in-field product sets are identified.

Example 4 (Figure 7.2): We illustrate this procedure using a GF(2^4) multiplier implemented using irreducible polynomial $P(x)_2$=x^4+x+1 (see Figure 7.1). Note that in this process, the labels do not offer any knowledge of the bit positions of inputs and outputs. The extracted algebraic expressions of the four output bits are shown in Figure 7.2. We first identify the unique products that include set s_0=a_0b_0 in algebraic expression of z_0; set s_1=(a_0b_1+a_1b_0) in z_1; set s_2=(a_0b_2+a_1b_1+a_2b_0) in z_2; and set s_3=(a_0b_3+a_1b_2+a_2b_1+a_3b_0) in z_3. Note that the number of products in the in-field product sets s_i is i. Hence, we have all the in-field product sets and their relation to the extracted algebraic expressions and make the the following conclusion:

- $s_0 = a_0b_0$, $z_0 \rightarrow$ Least significant bit (LSB)
- $s_1 = a_0b_1$+a_1b_0, $z_1 \rightarrow 2^{nd}$ output bit
- $s_2 = a_0b_2$+a_1b_1+a_2b_0, $z_2 \rightarrow 3^{rd}$ output bit
- $s_3 = a_0b_3$+a_1b_2+a_2b_1+a_3b_0, $z_3 \rightarrow$ Most significant bit (MSB)

7.4.2 Input encoding determination

We use Algorithm 2 to determine the bit position of the input variables. The input bit position can be determined by analyzing the in-field product sets, which has been obtained in the previous step. Based on the GF multiplication algorithm, we know that s_0 is generated by an AND function with two LSBs of the two inputs; and the two products in s_1 are generated by AND and XOR operations using two LSBs and two 2^{nd} input bits, etc. For example in a GF(2^4) multiplication (Figure 7.1), s_0=a_0b_0, where a_0 and b_0 are LSBs; s_1=a_1b_0+a_0b_1, where a_0, b_0 are LSBs; a_1, b_1 are 2^{nd} LSBs. This allows us to determine the bit position of the input bits recursively

Algorithm 5 Input encoding determination for $GF(2^m)$

Input: a set of algebraic expressions represent the in-field product sets S
Output: bit position of input variables

```
1: S={s_0, s_1, ..., s_{m-1}}
2: initialize a vector of variables V ← {}
3: for i=0, i≤m-1, i++ do
4:     for each variable v in algebraic expression of s_i do
5:         if v does not exist in V then
6:             assign bit position value of v = i
7:             store v in variable set V
8:         end if
9:     end for
10: end for
11: return V
```

by analyzing the algebraic expression of s_i. We illustrate our Algorithm 2 using the GF(2^4) multiplier implemented using $P(x)_2 = x^4+x+1$ (Figure 7.2).

Example 5 (Algorithm 2): The input of our algorithm is a set of algebraic expressions of the in-field product sets, s_0, s_1, s_2, s_3 (line 1). We initialize vector V to store the variables in which their bit positions are assigned (line 2). The first algebraic expression is s_0. Since the two variables, a_0 and b_0 are not in V, the bit positions of these two variables are assigned index $i = 0$ (line 4-8). In the second iteration, V=$\{a_0,\ b_0\}$, and the input algebraic expression is s_1, including variables a_0, b_0, a_1 and b_1. Because a_1 and b_1 are not in V, their bit position is $i = 1$. The loop ends when all the algebraic expressions in S have been visited, and returns V=$\{(a_0, b_0)_0,\ (a_1, b_1)_1,\ (a_2, b_2)_2,\ (a_3, b_3)_3\}$. The subscripts are the bit position values of the variables returned by the algorithm. Note that this procedure only gives the bit position of the input bits; the information of how the input words are constructed is unknown. There are 2^{m-1} combinations from which the words can be constructed using the information returned in V. For example, the two input words can be W_0=a_0+$2a_1$+$4b_2$+$8a_3$ and W_1=b_0+$2b_1$+$4a_2$+$8b_3$; or they can be W'_0=a_0+$2a_1$+$4b_2$+$8b_3$ and W'_1=b_0+$2b_1$+$4a_2$+$8a_3$. Although there may be many

combinations for constructing the input words, the specification of the GF(2^m) is unique.

7.4.3 Extraction of the Irreducible Polynomial

Theorem 5: *Given a multiplication in GF(2^m), let the first out-field product set be s_m. Then, the irreducible polynomial $P(x)$ includes x^m and x^i iff all products in the set s_m exist in the algebraic expression of the i^{th} output bits, where $i \leq m$.*

Proof: Based on the definition of field arithmetic, the polynomial basis representation of s_m is $s_m x^m$. To reduce s_m into elements in the range [0, $m-1$] (with m output bits), the field reductions are performed modulo irreducible polynomial $P(x)$ with highest degree m. Based on the definition of irreducible polynomial, $P(x)$ is either a trinomial or a pentanomial with degree m. Let $P(x) = x^m + P'(x)$. Then,

$$s_m x^m \ mod \ (x^m + P'(x)) \ = \ s_m P'(x)$$

Hence, if x^i exists in $P'(x)$, it also exists in $P(x)$.

□

Even though the input bit positions have been determined in the previous step, we cannot directly generate s_m since the combination of the input bits for constructing the input words is still unknown. In Example 5 (m=4), we can see that s_m=$\{a_1b_3, a_2b_2, a_3b_1\}$ when input words are W_0 and W_1; but s_m=$\{a_1a_3, a_2b_2, b_1b_3\}$ when inputs words are W_0' and W_1'. To overcome this limitation, we create a set of products s_m', which includes all the possible products that can be generated based on all input combinations. The set s_m' includes the *true* products, i.e., those that exist in the first out-field product set; and it also includes some *dummy* products. The dummy products are those that never appear in the resulting algebraic expressions. Hence, we first generate the set s_m' and eliminate the dummy products by searching the algebraic

expressions. After this, we obtain s_m. Then, we use s_m to extract the irreducible polynomial $P(x)$ using Algorithm 3.

Example 6: We illustrate the method of reverse engineering the irreducible polynomial using the $GF(2^4)$ multiplier of Fig. 1. The algorithm is shown in Algorithm 3. The extracted algebraic expressions S is shown in Figure 7.2 (line 1 at Algorithm 3). The bit position of input bits is determined by Algorithm 2 (line 2). Based on the result of Algorithm 2, we generate s'_m=$\{a_1a_3, b_1b_3, a_2b_2, a_3b_1, a_1b_3\}$. To eliminate the dummy products from s'_m, we search all algebraic expressions in S, and eliminate the products that cannot be a part of the resulting products. In this case, we find that a_1a_3 and b_1b_3 are the dummy products. Hence, we get s_m=$\{a_3b_1, a_2b_2, a_1b_3\}$ (line 3). Based on the definition of irreducible polynomial, $P(x)$ must include x^m; in this example $m = 4$ (line 4). While looping over all the algebraic expressions, the expressions for z_0 and z_1 contain all the products of s_m. Hence, x^0 and x^1 are included in $P(x)$, so that $P(x) = x^4+x^1+x^0$. We can see that it is the same as $P(x)_2$ in Figure 7.1.

Algorithm 6 Extracting irreducible polynomial in $GF(2^m)$

Input: the algebraic expressions of output bits S
Input: the first out-field product set s_m
Output: Irreducible polynomial $P(x)$

```
1: S = {exp_0, exp_1, ..., exp_{m-1}}
2: V ← Algorithm 2(S)
3: s_m ← eliminate_dummy(s'_m ← V, S)
4: P(x)=x^m: initialize irreducible polynomial
5: for i=0, i≤m-1, i++ do
6:     if all products in s_m exist in exp_i then
7:         P(x) += x^i
8:     end if
9: end for
10: return P(x)
```

In summary, using the framework presented in Section 3, we first extract the algebraic expressions of all output bits. Then, we analyze the algebraic expressions to

Table 7.1: Results of verifying Mastrovito multipliers using our parallel approach. T is the number of threads. MO=Memory out of 32 GB. TO=Time out of 12 hours. (*T=*1* shows the maximum memory usage of a single thread.)

Mastrovito		[91]		This work				
Op size	# equations	Runtime (sec)	Mem (MB)	Runtime (s)				Mem*
				T=5	*T=10*	*T=20*	*T=30*	*T=1**
32	5,482	1	3	2	1	1	1	10 MB
48	12,228	8	13	6	3	3	2	21 MB
64	21,814	29	21	11	8	7	7	37 MB
96	51,412	195	45	38	26	20	23	84 MB
128	93,996	924	91	91	63	55	57	152 MB
163	153,245	3546	161	192	137	121	113	248 MB
233	167,803	4933	168	294	212	180	171	270 MB
283	399,688	30358	380	890	606	550	530	642 MB
571	1628,170	*TO*	-	7980	5038	*MO*	*MO*	2.6 GB

Table 7.2: Results of verifying *Montgomery* multipliers using our parallel approach. T is the number of threads. TO=Time out of 12 hours. MO=Memory out of 32 GB. (*T=*1* shows the maximum memory usage of a single thread.)

Montgomery		[91]		This work				
Op size	# equations	Runtime (sec)	Mem (MB)	Runtime (s)				Mem*
				T=5	*T=10*	*T=20*	*T=30*	*T=1**
32	4,352	2	3	3	2	1	2	8 MB
48	9,602	14	13	18	11	9	6	16 MB
64	16.898	63	21	45	31	28	27	27 MB
96	37,634	554	45	234	157	133	142	59 MB
128	66,562	1924	68	209	121	115	110	95 MB
163	107,582	12063	101	1616	1172	1095	1008	161 MB
233	219,022	*TO*	168	722	565	457	480	301 MB
283	322,622	*TO*	380	19745	17640	15300	14820	488 MB

find the bit position of the input bits and the output bits, and extract the irreducible polynomial $P(x)$. In the example of the GF(2^4) multiplier implemented using $P(x)$ = x^4+x+1, shown in Figure 7.1, the final results returned by our approach gives the following: **1)** the input bits set V= $\{(a_0, b_0)_0$, $(a_1, b_1)_1$, $(a_2, b_2)_2$, $(a_3, b_3)_3\}$, where the subscripts represent the bit position; **2)** z_0 is the least significant bit (LSB), z_1 is the 2^{nd} output bit, z_2 is the 3^{rd} output bit, and z_3 is the most significant bit (MSB); **3)** irreducible polynomial is $P(x) = x^4$+x+1; **4)** the specification can be verified by applying the technique presented in Section 3 with the reverse engineered information.

7.5 Results

We present the results of our method in two subsections: 1) evaluation of parallel verification of GF(2^m) multipliers; and 2) evaluation of reverse engineering GF(2^m) multipliers.

7.5.1 Parallel Verification of GF(2^m) Multipliers

The verification technique for GF(2^m) multipliers presented in Section III, was implemented in C++. It performs backward rewriting with variable substitution and polynomial reductions in Galois field in parallel fashion. The program was tested on a number of combinational gate-level $GF(2^m)$ multipliers taken from [70], including the Montgomery multipliers [58] and Mastrovito multipliers [115]. The bit-width of the multipliers varies from 32 to 571 bits. The experiments of verifying Galois field multipliers using SAT, SMT, ABC [80] and Singular [40] have been already presented in [70] and [91]. They demonstrate that the rewriting technique performs significantly better than other known techniques. Hence, in this work, we only compare our approach to those of [70] and [91]. Specifically, we compare our approach to the tool described in [91]. In contrast to the work of [70] and [91], all the GF(2^m) multipliers used in this section are ***bit-blasted*** gate-level implementations. We take the bit-level multipliers from [91] and map them into gate-level circuits using ABC [80]. Our experiments were conducted on a PC with Intel(R) Xeon CPU E5-2420 v2 2.20 GHz $\times$12 with 32 GB memory. As described in the next section, our technique can verify Galois field multipliers in multiple threads (up to 30 using our platform). In each thread, Algorithm 1 is applied on a single output bit. The number of threads is given as input to the tool.

The experimental results of our approach and comparison against [91] are shown in Table 7.1 for gate-level Mastrovito multipliers with bit-width varying from 32 to 571 bits. These multipliers are directly mapped using ABC [80] without any optimization.

The largest circuit includes more than 1.6 million gates. This is also the number of polynomial equations and the number of rewriting iterations (see Section III). The results generated by the tool, presented in [91] are shown in columns 3 and 4. We performed four different series of experiments, with a different number of threads, T=5, 10, 20, and 30. The runtime results are shown in columns 6 to 8 and memory usage in column 9. The timeout limit (TO) was set to 12 hours and memory limit (MO) to 32 GB. The experimental results show that our approach provides on average 26.2×, 37.8×, 42.7×, and 44.3× speedup, for $T = 5$, 10, 20, and 30 threads, respectively. Our approach can verify the multipliers up to 571 bit-wide multipliers in 1.5 hours, while that of [91] fails after 12 hours.

Note that the reported memory usage of our approach is the maximum memory usage *per thread*. This means that our tool experiences maximum memory usage with all T threads running in the process; in this case, the memory usage is $T \cdot Mem$. This is why the 571-bit Mastrovito multipliers could be successfully verified with $T = 5$ and 10, but failed with $T = 20$ and 30 threads. For example, the peak memory usage of 571-bit Mastrovito multiplier with $T = 20$ is $2.6 \times 20 = 52$ GB, which exceeds the available memory limit.

We also tested Montgomery multipliers with bit-width varying from 32 to 283 bits. These experiments are different than those in [91]. In our work, we first flatten the Montgomery multipliers before applying our verification technique. That is, we assume that only the positions of the primary inputs and outputs are known, without the knowledge of the internal structure or clear boundaries of the blocks inside the design. The results are shown in Table 7.2. For 32- to 163-bit Montgomery multipliers, our approach provides on average a 9.2×, 15.9×, 16.6×, and 17.4× speedup, for $T = 5$, 10, 20, and 30, respectively. Notice that [91] cannot verify the flattened Montgomery multipliers larger than 233 bits in 12 hours.

One observation is that extracting polynomials expressions of Montgomery multipliers require more time than Mastrovito multipliers. The main reason causing the difference is the architecture of the multipliers. Mastrovito multipliers are directly implemented based on the multiplication structure, i.e., the partial product generator following by XOR-tree structures for modular arithmetic. Since the algebraic model of XOR is the simplest model (linear model) in GF arithmetic, the size of the polynomial expressions of rewriting of this architecture is small. In contrast, Montgomery multiplier will first transform the two integer inputs into Montgomery forms. The modular arithmetic is then applied on these two Montgomery forms. Note that the polynomial expressions of Montgomery forms are much larger than partial products, which increase the size of intermediate expressions.

7.5.1.1 Design and Verification cost depend on $P(x)$

In Table 7.2, we observe that CPU runtime for verifying a 163-bit multiplier is greater than that of a 233-bit multiplier. This is because the computation complexity depends not only on the bit-width of the multiplier, but also on the irreducible polynomial $P(x)$.

We illustrate this using two GF(2^4) multiplications implemented using two different irreducible polynomials (refer to Figure 7.1). We can see that when $P(x)_1$=$x^4 + x^3 + 1$, the longest logic paths for z_3 and z_0, include ten and seven products that need to be generated using XORs, respectively. However, when $P(x)_2$=$x^4 + x + 1$, the two longest paths, z_1 and z_2, have only seven and six products. This means that the GF(2^4) multiplication requires **9** XOR operations using $P_1(x)$ and requires **6** XOR operations using $P_2(x)$. In other words, the gate-level implementation of the multiplier implemented using $P_1(x)$ has more gates compared to $P_2(x)$. In conclusion, we can see that irreducible polynomial $P(x)$ has significant impact on both design cost and the verification time of the GF(2^m) multipliers.

7.5.1.2 Runtime vs. Memory of Parallelism

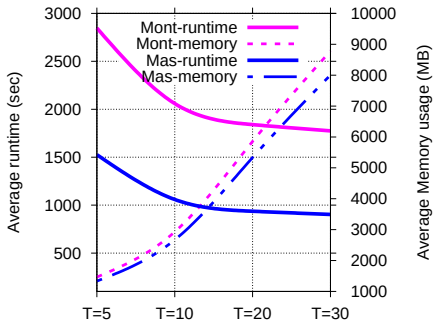


Figure 7.8: Runtime and memory usage of our parallel verification approach as a function of number of threads T.

In this section, we discuss the tradeoff of runtime and memory usage of our parallel approach to Galois Field multiplier verification. The plots in Figure 7.8 show how the average runtime and memory usage change with a different number of threads. The vertical axis on the left is CPU runtime (in seconds), and on the right is memory usage (MB). The horizontal axis represents the number of threads T, ranging from 5 to 30. The runtime is significantly improved for T ranging from 5 and 15. However, there is not much speedup when T is greater than 20, most likely due to the memory management synchronization overhead between the threads. Based on the experiments of Mastrovito multipliers (Table 7.1), our approach is limited by the memory usage when the size of the multiplier and T are large. In our work, $T = 20$ seems to be the best choice. Obviously, T varies between the platforms depending on the number of cores and the memory.

In addition to analyze the runtime complexity of our rewriting algorithm, an analysis using single thread (i.e. T=1) is shown in Figure 7.9. The y-axis shows the runtime of extracting the polynomial expressions, and x-axis indicates the size of the Mastrovitor Multipliers. The result shows that the runtime complexities using different T are almost the same.

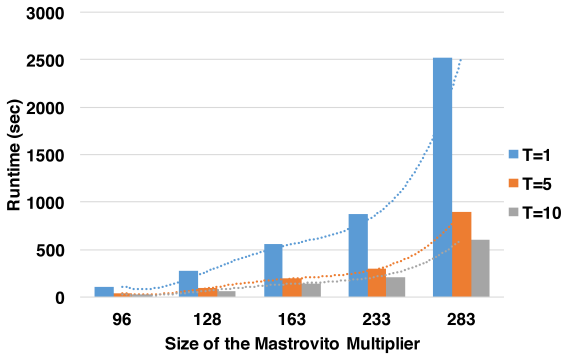


Figure 7.9: Sing thread runtime analysis using Mastrovito multipliers.

7.5.1.3 Effect of synthesis on verification of GF(2^m) multipliers

We conclude that highly bit-optimized integer arithmetic circuits are harder to verify than their original, pre-synthesized netlists[125]. This is because the efficiency of the rewriting technique relies on the amount of cancellations between the different terms of the polynomial, and the cancellations strongly depend on the order in which signals are rewritten. A good ordering of signals is difficult to be achieved in highly bit-optimized circuits.

To see the effect of synthesis on parallel verification of GF circuits, we applied our approach to *post-synthesized* Galois field multipliers with operands up to 409 bits (571-bit multipliers could not be synthesized in a reasonable time). We synthesized *Mastrovito* and *Montgomery* multipliers using *ABC* tool [80]. We repeatedly used the commands *resyn2* and *dch*[2] until *ABC* could not reduce the number of levels or the number of nodes anymore. The synthesized multipliers were mapped using a 14nm technology library. The verification experiments shown in Table 7.3 are performed by our tool with $T = 20$ threads. Our tool was able to verify both 409-bit *Mastrovito* and *Montgomery* multipliers within just 13 minutes. We observe that the Galois field multipliers are much easier to be verified after optimization using our parallel approach. For example, the verification of a 283-bit Montgomery multiplier takes

[2]"dch" is the most efficient bit-optimization function in ABC.

15,300 seconds when T =20. After optimization, the runtime was just 169.2 seconds, which is 90x faster than verifying the original implementation. The memory usage is also reduced from 488 MB to 194 MB. In summary, in contrast to verification problems of integer multipliers [125], the bit-level optimization actually reduces the complexity of backward rewriting process. This is because extracting the function of an output bit of a GF multiplier depends only on the logic cone of that bit and does not require logic from other bits to be simplified (see Theorem 3). Hence, the complexity of function extraction is naturally reduced if the logic cone is minimized.

Table 7.3: Runtime and memory usage of synthesized *Mastrovito* and *Montgomery* multipliers (T=20).

Op size	*Mastrovito*		*Montgomery*	
	Runtime(s)	Mem	Runtime(s)	Mem
64	4	21 MB	15	38 MB
96	11	44 MB	41	54 MB
128	29	77 MB	27	78 MB
163	62	123 MB	205	153 MB
233	135	201 MB	141	199 MB
283	168	198 MB	169	194 MB
409	776	635 MB	751	597 MB

7.5.2 Reverse Engineering of GF(2^m) Multipliers

The reverse engineering technique presented in this chapter was implemented in the framework described in Section IV in C++. It reverse engineers bit-blasted GF(2^m) multipliers by analyzing the algebraic expressions of each element using the approach presented in Section III. The program was tested on a number of gate-level $GF(2^m)$ multipliers with different irreducible polynomials, including Montgomery multipliers and Mastrovito multipliers. The multiplier generator is taken from [70], takes the bit-width and the irreducible polynomial as inputs and generates the multipliers in equation or BLIF format. The experimental results show that our technique can successfully reverse engineer various of GF(2^m) multipliers, regardless of the GF(2^m) algorithm and the irreducible polynomials. The experiments Ire conducted on the PC with Intel(R) Xeon CPU E5-2420 v2 2.20 GHz ×12 with 32 GB

memory. The number of threads is set to 16 for all the reverse engineering evaluations in this section. This is dictated by the fact that T=16 gives most promising performance (runtime) and scalability (memory usage) on our platform, based on the analysis presented in Section V-A (Figure 7.8).

Table 7.4: Results of reverse engineering synthesized and technology mapped Mastrovito and Montgomery multipliers.

m	$P(x)$	*Mastrovito-syn*		*Montgomery-syn*	
		Runtime(s)	Mem	Runtime(s)	Mem
64	$x^{64}+x^{21}+x^{19}+x^4+1$	13	25 MB	5	20 MB
163	$x^{163}+x^{80}+x^{47}+x^9+1$	69	508 MB	221	610 MB
233	$x^{233}+x^{74}+1$	152	1.2 GB	154	2.9 GB
409	$x^{409}+x^{87}+1$	825	6.5 GB	855	10.3 GB

Our program takes the netlist/equations of the GF(2^m) implementations, and the number of threads as input. Hence, the users can adjust the parallel efforts depending on the limitation of the machines. In this work, all results are performed in 16 threads. Typical designs that require reverse engineering are those that have been bit-optimized and mapped using a standard-cell library. Hence, we apply our technique to the bit-optimized Mastrovito and Montgomery multipliers (Table 7.4). For the purpose of our experiments, the multipliers are optimized and mapped using ABC [80]. Compared to the runtime of verifying synthesized multipliers (Table 7.3), the runtime spent on analyzing the extracted expressions for reverse engineering is less than 10% of extraction process. This is because most computations of reverse engineering approach are those for extracting the algebraic expressions, as presented in Section V-A. Table 7.3.

This approach is further evaluated using four Mastrovito multipliers implemented with four irreducible polynomials in the same field GF(2^{233}). Those polynomials are obtained from [102]. The results are shown in Figure 7.10 and those multipliers are optimized using ABC synthesis. We can see that the multipliers implemented with trinomial $P(x)$ are much easier to be reverse engineered than pentanomial $P(x)$.

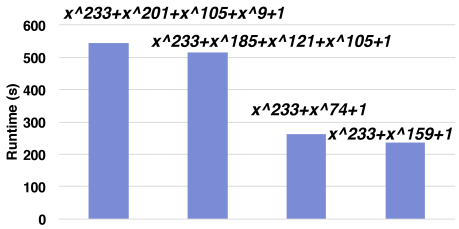


Figure 7.10: Result of reverse engineering GF(2^{233}) Mastrovito multipliers that are implemented using different P(x).

This is because the multipliers implemented with pentanomial P(x) contain much more gates, and have longer critical path, since the reduction over pentanomial requires much more XOR operations. The runtime between different trinomials or pentanomials are almost the same. Comparing the design efficiency between the two trinomials, the efficient trinomial irreducible polynomial, x^m+x^a+1, mostly satisfies m-$a>m/2$. The results of area and critical path delay after logic synthesis and technology mapping with 14nm technology library are shown in Figure 7.11. It shows that the area and delay of the Mastrovito multiplier implemented with $P(x)=x^{233}+x^{74}+1$ are 5.7% and 7.4% less than $P(x)=x^{233}+x^{159}+1$.

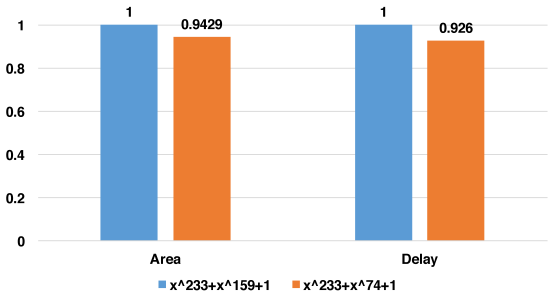


Figure 7.11: Evaluation of the design cost using GF($2^2$33) Mastrovito multipliers with irreducible polynomials $x^{233}+x^{159}+1$ and $x^{233}+x^{74}+1$.

7.6 Conclusion

This Chapter presents a parallel approach to verification and reverse engineering of gate-level Galois Field multipliers using computer algebraic methods. It introduces a parallel rewriting method that can efficiently extract polynomial expressions of Galois

Field multipliers. We demonstrate that, compared to the best existing algorithms, our approach tested on T=30 threads, provides average 44× and 17× speedup for verification of Montgomery and Mastrovito multipliers, respectively.

Based on the proposed parallel rewriting technique, we presented a novel approach that reverse engineers the gate-level Galois Field multipliers, in which the irreducible polynomial, as well as the bit position of the inputs and outputs are unknown. We show that our approach can efficiently reverse engineer the Galois Field multipliers implemented using different irreducible polynomials. Future work will focus on formal verification of prime field arithmetic circuits and complex cryptography circuits.

BIBLIOGRAPHY

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

[2] Amla, Nina, Du, Xiaoqun, Kuehlmann, Andreas, Kurshan, Robert P, and McMillan, Kenneth L. An analysis of sat-based model checking techniques in an industrial environment. In *Correct hardware design and verification methods*. Springer, 2005, pp. 254–268.

[3] Andraus, Zaher S, Liffiton, Mark H, and Sakallah, Karem A. Refinement Strategies for Verification methods based on Datapath abstraction. In *ASP-DAC 2006* (2006), IEEE Press, pp. 19–24.

[4] Andraus, Zaher S, and Sakallah, Karem A. Automatic Abstraction and Verification of Verilog Models. In *Proceedings of the 41st annual Design Automation Conference* (2004), ACM, pp. 218–223.

[5] Ashar, Pranav, Ghosh, Abhijit, and Devadas, Srinivas. Boolean satisfiability and equivalence checking using general binary decision diagrams. *Integration, the VLSI journal 13*, 1 (1992), 1–16.

[6] Barrett, Clark, Conway, Christopher L, Deters, Morgan, Hadarean, Liana, Jovanović, Dejan, King, Tim, Reynolds, Andrew, and Tinelli, Cesare. CVC4. In *Computer aided verification (CAV)* (2011), Springer, pp. 171–177.

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

[8] Baumgarten, Alex, Tyagi, Akhilesh, and Zambreno, Joseph. Preventing ic piracy using reconfigurable logic barriers. *IEEE Design & Test of Computers 27*, 1 (2010).

[9] Beatty, Derek L, Bryant, Randal E, and Seger, Carl-Johan H. Synchronous circuit verification by symbolic simulation: an illustration. In *Proceedings of the sixth MIT conference on Advanced research in VLSI* (1990), MIT Press, pp. 98–112.

[10] Beers, Robert. Pre-Rtl Formal Verification: an Intel Experience. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE* (2008), IEEE, pp. 806–811.

[11] Belov, Anton, Diepold, Daniel, Heule, Marijn JH, and Järvisalo, Matti. SAT Competition 2014. *SAT* (2014).

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

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

[14] Biere, Armin, Cimatti, Alessandro, Clarke, Edmund M, Fujita, Masahiro, and Zhu, Yunshan. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference* (1999), ACM, pp. 317–320.

[15] Biere, Armin, Cimatti, Alessandro, Clarke, Edmund M, Strichman, Ofer, and Zhu, Yunshan. Bounded model checking. *Advances in computers 58* (2003), 117–148.

[16] Biere, Armin, Heule, Marijn, and van Maaren, Hans. *Handbook of satisfiability*, vol. 185. ios press, 2009.

[17] Bombieri, Nicola, Fummi, Franco, Guarnieri, Valerio, Pravadelli, Graziano, and Vinco, Sara. Redesign and verification of rtl ips through rtl-to-tlm abstraction and tlm synthesis. In *Microprocessor Test and Verification (MTV), 2012 13th International Workshop on* (2012), IEEE, pp. 76–81.

[18] Bose, Soumitra, and Fisher, Allan L. Verifying pipelined hardware using symbolic logic simulation. In *Computer Design: VLSI in Computers and Processors, 1989. ICCD'89. Proceedings., 1989 IEEE International Conference on* (1989), IEEE, pp. 217–221.

[19] Brock, Bishop, Kaufmann, Matt, and Moore, J Strother. Acl2 theorems about commercial microprocessors. In *Formal Methods in Computer-Aided Design (FMCAD)* (1996), Springer, pp. 275–293.

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

[21] Bryant, Randal E. Symbolic simulationtechniques and applications. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (1991), ACM, pp. 517–521.

[22] Bryant, Randal E., and Chen, Yirng-An. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the 32st Conference on Design Automation, San Francisco, California, USA, Moscone Center, June 12-16, 1995.* (1995), pp. 535–541.

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

[24] Burch, Jerry R. Using bdds to verify multipliers. In *Proceedings of the 28th ACM/IEEE Design Automation Conference* (1991), ACM, pp. 408–412.

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

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

[27] Chen, Yirng-An, Clarke, Edmund, Ho, Pei-Hsin, Hoskote, Yatin, Kam, Timothy, Khaira, Manpreet, O'Leary, John, and Zhao, Xudong. Verification of all circuits in a floating-point unit using word-level model checking. In *Formal Methods in Computer-Aided Design* (1996), Springer, pp. 19–33.

[28] Ciesielski, M., Gomez-Prado, D., Ren, Q., Guillot, J., and Boutillon, E. Optimization of Data-Flow computation using Canonical TED Representation. *IEEE Trans. on Computers 28*, 9 (September 2009), 1321–1333.

[29] Ciesielski, M., Kalla, P., and Askar, S. Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs. *IEEE Trans. on Computers 55*, 9 (Sept. 2006), 1188–1201.

[30] 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.

[31] Ciesielski, M, Yu, C, Brown, W, Liu, D, and Rossi, André. Verification of Gate-level Arithmetic Circuits by Function Extraction. In *52nd DAC* (2015), ACM, pp. 52–57.

[32] Ciesielski, Maciej, Brown, Walter, Liu, Duo, and Rossi, André. Function extraction from arithmetic bit-level circuits. In *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on* (2014), IEEE, pp. 356–361.

[33] Ciet, Mathieu, Quisquater, Jean-Jacques, and Sica, Francesco. A short note on irreducible trinomials in binary fields. In *23rd Symposium on Information Theory in the BENELUX* (2002).

[34] Cimatti, Alessandro, Clarke, Edmund, Giunchiglia, Fausto, and Roveri, Marco. Nusmv: A new symbolic model verifier. In *Computer Aided Verification* (1999), Springer, pp. 495–499.

[35] Clarke, Edmund, Grumberg, Orna, Jha, Somesh, Lu, Yuan, and Veith, Helmut. Counterexample-guided Abstraction Refinement. In *CAV 2000* (2000), Springer, pp. 154–169.

[36] Clarke, Edmund M, Grumberg, Orna, and Peled, Doron. *Model checking*. MIT press, 1999.

[37] Cohen, Aaron E, and Parhi, Keshab K. Architecture optimizations for the rsa public key cryptosystem: a tutorial. *Circuits and Systems Magazine, IEEE 11*, 4 (2011), 24–34.

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

[39] De Moura, Leonardo, and Bjørner, Nikolaj. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[40] 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.

[41] El Massad, Mohamed, Garg, Siddharth, and Tripunitara, Mahesh V. Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes. In *NDSS* (2015).

[42] Fallah, Farzan, Devadas, Srinivas, and Keutzer, Kurt. Functional vector generation for hdl models using linear programming and 3-satisfiability. In *Design Automation Conference (DAC)* (1998), IEEE, pp. 528–533.

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

[44] Foster, Harry D. Trends in functional verification: A 2014 industry study. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE* (2015), IEEE, pp. 1–6.

[45] Ghandali, Samaneh, Alizadeh, Bijan, Fujita, Masahiro, and Navabi, Zainalabedin. Automatic high-level data-flow synthesis and optimization of polynomial datapaths using functional decomposition. *Computers, IEEE Transactions on* (2014).

[46] Ghandali, Samaneh, Yu, Cunxi, Liu, Duo, Walter, Brown, , and Ciesielski, Maciej. Logic Debugging of Arithmetic Circuits. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2015), IEEE, pp. 113–118.

[47] Goldberg, Evgueni, Prasad, Mukul, and Brayton, Robert. Using sat for combinational equivalence checking. In *Proceedings of the conference on Design, automation and test in Europe* (2001), IEEE Press, pp. 114–121.

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

[49] Hansen, Mark C, Yalcin, Hakan, and Hayes, John P. Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering. *IEEE Design & Test of Computers*, 3 (1999), 72–80.

[50] Himanshu, Jain, and et al. Word-level Predicate-abstraction and Refinement Techniques for Verifying RTL verilog. *TCAD* (2008).

[51] Huang, Zheng, Wang, Lingli, Nasikovskiy, Yakov, and Mishchenko, Alan. Fast boolean matching based on NPN classification. In *International Conference on Field-Programmable Technology, FPT, Kyoto, Japan* (2013).

[52] Jain, Himanshu, Kroening, Daniel, Sharygina, Natasha, and Clarke, Edmund. Word level Predicate Abstraction and Refinement for Verifying RTL verilog. In *42nd DAC* (2005), ACM, pp. 445–450.

[53] Kaiss, Daher, Skaba, Marcelo, Hanna, Ziyad, and Khasidashvili, Zurab. Industrial strength sat-based alignability algorithm for hardware equivalence verification. In *FMCAD* (2007), IEEE, pp. 20–26.

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

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

[56] Kaufmann, Matt, and Moore, J Strother. Acl2: An industrial strength version of nqthm. In *COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on Computer Assurance* (1996), IEEE, pp. 23–34.

[57] Keutzer, K. Dagon: Technology binding and local optimization by dag matching.

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

[59] Koelbl, Alfred, Jacoby, Reily, Jain, Himanshu, and Pixley, Carl. Solver technology for system-level to rtl equivalence checking. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.* (2009), IEEE, pp. 196–201.

[60] Koelbl, Alfred, and Pixley, Carl. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming 33*, 6 (2005), 645–666.

[61] Koren, Israel. *Computer Arithmetic Algorithms*. Universities Press, 2002.

[62] Krautz, Udo, Paruthi, Viresh, Arunagiri, Anand, Kumar, Sujeet, Pujar, Shweta, and Babinsky, Tina. Automatic verification of floating point units. In *Proceedings of the 51st Annual Design Automation Conference* (2014), ACM, pp. 1–6.

[63] Kuehlmann, Andreas, and Krohm, Florian. Equivalence checking using cuts and heaps. In *Proceedings of the 34th annual Design Automation Conference* (1997), ACM, pp. 263–268.

[64] Kuehlmann, Andreas, Srinivasan, Arvind, and LaPotin, David P. Verity-a formal verification program for custom cmos circuits. *IBM Journal of Research and Development 39*, 1-2 (1995), 149–166.

[65] Li, Meng, Shamsi, Kaveh, Meade, Travis, Zhao, Zheng, Yu, Bei, Jin, Yier, and Pan, David Z. Provably secure camouflaging strategy for IC protection. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016* (2016), p. 28.

[66] Li, Wenchao, Gascon, Adria, Subramanyan, Pramod, Tan, Wei Yang, Tiwari, Anish, Malik, Sharad, Shankar, Nishanth, Seshia, Sanjit, et al. Wordrev: Finding Word-level Structures in a Sea of Bit-level Gates. In *HOST 2013* (2013), IEEE, pp. 67–74.

[67] Li, Wenchao, Wasson, Zach, Seshia, Sanjit, et al. Reverse Engineering Circuits using Behavioral Pattern Mining. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on* (2012), IEEE, pp. 83–88.

[68] Li, Wenchao, Wasson, Zach, and Seshia, Sanjit A. Reverse engineering circuits using behavioral pattern mining. In *2012 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2012, San Francisco, CA, USA, June 3-4, 2012* (2012), pp. 83–88.

[69] Liu, Duo, Yu, Cunxi, Zhang, Xiangyu, and Holcomb, Daniel E. Oracle-guided incremental SAT solving to reverse engineer camouflaged logic circuits. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016* (2016), pp. 433–438.

[70] Lv, J., Kalla, P., and Enescu, F. Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmatic Circuits. *IEEE Trans. on CAD 32*, 9 (September 2013), 1409–1420.

[71] Lv, J., Kalla, P., and Enescu, F. Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits. *IEEE Trans. on CAD 32*, 9 (September 2013), 1409–1420.

[72] Lv, Jinpeng, Kalla, Priyank, and Enescu, Florian. Efficient groebner basis reductions for formal verification of galois field arithmetic circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 32*, 9 (2013), 1409–1420.

[73] Manolios, Panagiotis, and Srinivasan, Sudarshan K. Refinement Maps for Efficient Verification of Processor Models. In *DATE 2005* (2005), pp. 1304–1309.

[74] Mishchenko, A., Een, N., Brayton, R. K., Jang, S., Ciesielski, M., and Daniel, T. MAGIC: An Industrial-Strength Logic Optimization, Technology Mapping, and Formal Verification Tool. In *Intl. Workshop on Logic Synthesis* (June 2010), pp. 124–127.

[75] Mishchenko, A, et al. ABC: A System for Sequential Synthesis and Verification. *URL http://www. eecs. berkeley. edu/˜ alanmi/abc* (2007).

[76] Mishchenko, Alan, Chatterjee, Satrajit, and Brayton, Robert. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd annual Design Automation Conference* (2006), ACM, pp. 532–535.

[77] Mishchenko, Alan, Chatterjee, Satrajit, and Brayton, Robert. DAG-aware AIG Rewriting A Fresh Look at Combinational Logic Synthesis. In *43rd DAC* (2006), ACM, pp. 532–535.

[78] Mishchenko, Alan, Chatterjee, Satrajit, Jiang, Roland, and Brayton, Robert K. Fraigs: A unifying representation for logic synthesis and verification. Tech. rep., ERL Technical Report, 2005.

[79] Mishchenko, Alan, Chatterjee, Satrajit, Jiang, Roland, and Brayton, Robert K. Fraigs: A unifying representation for logic synthesis and verification. Tech. rep., ERL Technical Report, 2005.

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

[81] Naoyuki, Tamura, Takehide, Soh, and Mutsunori, Banbara. PBSugar: A SAT-based Pseudo-Boolean Solver. *http://bach.istc.kobe-u.ac.jp/pbsugar* (2013).

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

[83] NIST. Recommended elliptic curves for federal government use.

[84] Owre, Sam, Rushby, John M, and Shankar, Natarajan. PVS: A Prototype Verification System. In *Automated Deduction - CADE-11*. Springer, 1992, pp. 748–752.

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

[86] Pan, Peichen, and Lin, Chih-Chang. A new retiming-based technology mapping algorithm for lut-based fpgas. In *FPGA* (1998), pp. 35–42.

[87] Parthasarathy, Ganapathy, Huang, Chung-Yang, and Cheng, Kwang-Ting. An Analysis of ATPG and SAT Algorithms for Formal Verification. In *HLDVT. Proceedings. Sixth IEEE International* (2001), IEEE, pp. 177–182.

[88] Paruthi, Viresh, and Kuehlmann, Andreas. Equivalence checking combining a structural sat-solver, bdds, and simulation. In *Computer Design, 2000. Proceedings. 2000 International Conference on* (2000), IEEE, pp. 459–464.

[89] 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.

[90] Plaza, Stephen M, Chang, Kai-hui, Markov, Igor L, and Bertacco, Valeria. Node mergers in the presence of don't cares. In *ASP-DAC'07* (2007), IEEE, pp. 414–419.

[91] 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.

[92] Pruss, Tim, Kalla, Priyank, and Enescu, Florian. Efficient symbolic computation for word-level abstraction from combinational circuits for verification over finite fields. *IEEE Trans. on CAD of Integrated Circuits and Systems 35*, 7 (2016), 1206–1218.

[93] Rajendran, Jeyavijayan, Pino, Youngok, Sinanoglu, Ozgur, and Karri, Ramesh. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference* (2012), ACM, pp. 83–89.

[94] Rajendran, Jeyavijayan, Sam, Michael, Sinanoglu, Ozgur, and Karri, Ramesh. Security analysis of integrated circuit camouflaging. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), pp. 709–720.

[95] Rajendran, Jeyavijayan, Zhang, Huan, Zhang, Chi, Rose, Garrett S, Pino, Youngok, Sinanoglu, Ozgur, and Karri, Ramesh. Fault analysis-based logic encryption. *IEEE Transactions on computers 64*, 2 (2015), 410–424.

[96] Raudvere, Tarvo, Singh, Ashish Kumar, Sander, Ingo, and Jantsch, Axel. System Level Verification of Digital Signal Processing Applications based on the Polynomial Abstraction Technique. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design (ICCAD)* (2005), IEEE Computer Society, pp. 285–290.

[97] Saab, Daniel G, Abraham, Jacob A, and Vedula, Vivekananda M. Formal Verification using Bounded Model Checking: SAT versus Sequential ATPG Engines. In *VLSI Design, 2003. Proceedings. 16th International Conference on* (2003), IEEE, pp. 243–248.

[98] SAT. Sat competition 2014. *URL http://www.satcompetition.org/2014/results.shtml* (2014 July 14-17 in Vienna, Austria), 53.

[99] Savoj, Hamid, Mishchenko, Alan, and Brayton, Robert. Sequential equivalence checking for clock-gated circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 33*, 2 (2014), 305–317.

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

[101] Sayed-Ahmed, Amr, Große, Daniel, Soeken, Mathias, and Drechsler, Rolf. Equivalence checking using grobner bases. *FMCAD'2016* (2016).

[102] Scott, Michael. Optimal irreducible polynomials for gf (2m) arithmetic. *IACR Cryptology ePrint Archive 2007* (2007), 192.

[103] Shekhar, N., Kalla, P., and Enescu, F. Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing. *IEEE Trans. on Computer-Aided Design 26*, 7 (July 2007), 1320–1330.

[104] Shekhar, Namrata, Kalla, Priyank, Enescu, Florian, and Gopalakrishnan, Sivaram. Exploiting vanishing polynomials for equivalence verification of fixed-size arithmetic datapaths. In *ICCD* (2005), IEEE, pp. 215–220.

[105] Soeken, Mathias, Sterin, Baruch, Drechsler, Rolf, and Brayton, Robert. Simulation Graphs for Reverse Engineering. *FMCAD 2015*.

[106] Sohofi, Hassan, and Navabi, Zainalabedin. Assertion-based verification for system-level designs. In *Quality Electronic Design (ISQED), 2014 15th International Symposium on* (2014), IEEE, pp. 582–588.

[107] Somenzi, Fabio. CUDD: CU Decision Diagram Package-release 2.4. 0. *University of Colorado at Boulder* (2009).

[108] Soos, Mate. Enhanced Gaussian Elimination in DPLL-based SAT Solvers. In *POS@ SAT* (2010), pp. 2–14.

[109] Sorensson, Niklas, and Een, Niklas. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT 2005* (2005), 53.

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

[111] Su, Tiankai, Yu, Cunxi, Yasin, Atif, and Ciesielski, Maciej J. Formal verification of truncated multipliers using algebraic approach and re-synthesis. In *2017 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2017, Bochum, Germany, July 3-5, 2017* (2017), pp. 415–420.

[112] Subramanyan, Pramod, Ray, Sayak, and Malik, Sharad. Evaluating the security of logic encryption algorithms. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015* (2015), pp. 137–143.

[113] Subramanyan, Pramod, Tsiskaridze, Nestan, Pasricha, Kanika, Reisman, Dillon, Susnea, Adriana, and Malik, Sharad. Reverse Engineering Digital Circuits Using Functional Analysis. In *DATE* (2013), pp. 1277–1280.

[114] Sun, Xiaojun, Kalla, Priyank, Pruss, Tim, and Enescu, Florian. Formal verification of sequential galois field arithmetic circuits using algebraic geometry. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2015), EDA Consortium, pp. 1623–1628.

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

[116] Synopsys. Synopsys Formality.

[117] Theo, Drane, and Himanshu, Jain. Formal Verification and Validation of High-level Optimizations of Arithmetic Datapath Blocks. In *SNUG Awards 2011* (2011), Synopsys.

[118] Torrance, Randy, and James, Dick. The state-of-the-art in IC Reverse Engineering. In *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 2009, pp. 363–381.

[119] Torrance, Randy, and James, Dick. The state-of-the-art in semiconductor reverse engineering. In *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011* (2011), pp. 333–338.

[120] Van Eijk, CAJ. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 19*, 7 (2000), 814–819.

[121] 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.

[122] 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.

[123] Yasin, Muhammad, Mazumdar, Bodhisatwa, Sinanoglu, Ozgur, and Rajendran, Jeyavijayan. Camoperturb: secure IC camouflaging for minterm protection. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016* (2016), p. 29.

[124] Yu, Cunxi, Brown, Walter, and Ciesielski, Maciej. Verification of Arithmetic Datapath Designs using Word-level Approach A Case Study. *ISCAS 2015 10*, 11.

[125] Yu, Cunxi, Brown, Walter, Liu, Duo, Rossi, André, and Ciesielski, Maciej J. Formal verification of arithmetic circuits using function extraction. *IEEE Trans. on CAD of Integrated Circuits and Systems 35*, 12 (2016), 2131–2142.

[126] Yu, Cunxi, Choudhury, Mihir, Sullivan, Andrew, and Ciesielski, Maciej J. DAG-aware Logic Synthesis of Datapaths. In *2017 International Conference On Computer Aided Design,ICCAD 2017, Irvine, CA, USA, November 13-16, 2017* (2017).

[127] Yu, Cunxi, and Ciesielski, Maciej. Formal Verification using Don't-care and Vanishing Polynomials. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2016), IEEE, pp. 284–289.

[128] Yu, Cunxi, and Ciesielski, Maciej J. Analyzing imprecise adders using bdds - A case study. In *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2016, Pittsburgh, PA, USA, July 11-13, 2016* (2016), pp. 152–157.

[129] Yu, Cunxi, and Ciesielski, Maciej J. Automatic word-level abstraction of datapath. In *IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016* (2016), pp. 1718–1721.

[130] Yu, Cunxi, Ciesielski, Maciej J., Choudhury, Mihir, and Sullivan, Andrew. Dag-aware logic synthesis of datapaths. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016* (2016), pp. 135:1–135:6.

[131] Yu, Cunxi, Holcomb, Daniel, and Ciesielski, Maciej. Reverse engineering of irreducible polynomials in gf(2^m) arithmetic. In *DATE 2017, Lausanne, Switzerland* (2017).

[132] Yu, Cunxi, Zhang, Xiangyu, Liu, Duo, Ciesielski, Maciej, and Holcomb, Daniel. Incremental sat-based reverse engineering of camouflaged logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).

[133] Zhu, Qi, Kitchen, Nathan, Kuehlmann, Andreas, and Sangiovanni-Vincentelli, Alberto. Sat sweeping with local observability don't-cares. In *Proceedings of the 43rd annual Design Automation Conference* (2006), ACM, pp. 229–234.