Highly Automated Formal Verification of Arithmetic Circuits

PhD thesis

AMR SAYED-AHMED

Highly Automated Formal Verification of Arithmetic Circuits

Amr Sayed-Ahmed University of Bremen

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Engineering
- Dr.-Ing. -

University of Bremen, Faculty 3
Department of Mathematics and Computer Science
Bremen, December 2016

Supervisory Committee

1. Prof. Dr. Rolf Drechsler (University of Bremen, Germany)
2. Prof. Dr. Christoph Scholl (University of Freiburg, Germany)

To my Son Abdoallah

ACKNOWLEDGMENTS

I would like to start by thanking my advisor Rolf Drechsler for his continuous support and guidance. I deeply appreciate the trust in me to be part of his research group, which allows me to learn and research about various interesting topics. I take this opportunity to express gratitude to my former advisor Hossam Fahmy, he sets my mind to become a researcher, I learned from him fruitful lessons that gave me the power to complete my PhD journey. This thesis would not have been possible without the help of my co-authors, Daniel Große, Ulrich Kühne, and Mathias Soeken, who support me with their academic knowledge, valuable feedback, and many insightful discussions and suggestions. I would also like to thank my committee members and my external examiner Christoph Scholl for their time and their constructive comments.

Words can not express how grateful I am to all members of my family, special thanks to my parents Magda and Abdelfatah as well as my wife Gylan for their constant encouragement and endless patience. Furthermore, the last four years would not have been even half as enjoyable without all my friends in Germany; nice discussions with Nahla Galal, Nabila Abdessaied, Judith Peters, and Ahmed AbdelMonem Mohamed have contributed immensely to my personality.

—Amr Sayed-Ahmed, December 2016

CONTENTS

1 INTRODUCTION

Verifying the functional correctness of integrated circuits is essential to provide high quality systems. Traditionally, industrial designs are validated by simulation, often using specialized test case generators [45, 51, 84, 104] to target specific areas. While such approaches are efficient at exposing bugs, they are inherently incomplete and cannot achieve a full coverage—evaluation of all input combinations. In many cases, this intelligent dynamic simulation leaves considerable doubts about the correctness of integrated circuits. This has motivated the development of *formal verification* techniques to provide a full-functional verification coverage and prove the consistency of the circuits with their functional specification.

Microprocessors are being used for safety-critical systems such as air planes, nuclear reactor controllers and medical instrumentations. Hence, formal verification for microprocessors has a vital role. There are also commercial arguments for verifying formally microprocessors, because of the cost of a recall, redesign, and refabrication. A well known example is the FDIV bug in Intel's Pentium processor [89] which costs nearly half a billion dollar in 1995. Since this famous bug, a considerable research effort has been spent developing automated and formal techniques which can prove the correctness of microprocessor designs beyond mere testing.

FORMAL VERIFICATION IN HARDWARE DESIGN

Formal verification methods are applied in different phases of the circuit's design flow, they benefit significantly the *design process* by uncovering early the design errors. Typically, the design process proceeds in several roughly defined phases, from *specification* (conceptual design), through *Transaction Level Modeling* (TLM), *Register Transfer Level* (RTL), synthesis of a *gate-level netlist*, and finally a *structural layout* as an input to the manufacture process. The design specification defines the functional of the design and the nonfunctional requirements such as clock rate or power consumption. This specification is implemented manually or automatically, resulting in the RTL description. *Hardware Description Languages* (HDLs)—such as VHDL and Verilog—allow the

description of the design at RTL. TLM supports an early evaluation of the overall performance of the design, where components can easily be replaced before working on the actual RTL implementation. To obtain a gate-level netlist from the RTL description, automatic synthesis tools are utilized. In order to manufacture the final circuit, the gate-level netlist is further transformed into a structural layout by placing and wiring the netlist gates, using highly optimized tools.

Formal verification constructs rigorous mathematical proofs to verify the correctness of the *Design Under Verification* (DUV). This thesis focuses on designs that are described at RTL or gate-level. The formal proofs are hard—if not impossible—to be done by hand due to their complex functionality. Typically, they are performed with the help of automated software tools. This motivates the developments of software-assisted verification methods which formulate the specification of the DUV as proof obligations and verify that the DUV meets these obligations via an algorithmic proof. This proof is performed by answering decision problems by 'yes' or 'no' via *decision procedures*. A naïve proof would formulate the verification problem as a single decision problem, in many cases, this proof is computationally infeasible since the complexity of the decision problem overruns the computational capabilities of the state-of-the-art decision procedures. This motivates the developing of formal verification techniques for the verification of complex industrial designs by deriving—manually or mechanically—proofs consisting of sets of non-complex problems that can be answered by classical decision procedures.

Theorem proving, *equivalence checking*, and *model checking*, are the state-of-the-art techniques that verify formally microprocessors designs. Interactive theorem proving [49, 52] (or theorem proving for short) can be simply thought of as a formal proof that is checked by a computer. This mathematical proof is aided in the creation and the checking by software tools called theorem provers such as HOL-Light [48], Coq [29], PVS [75] and ACL2 [59] which provide trusted mathematical knowledge in a form of large numbers of basic theories and lemmas. To check a proof that asserts the correctness of the DUV with respect to the specification, a theorem prover applies powerful deductive reasoning based on its mathematical knowledge and lemmas that build the proof. The correctness of each lemma in the proof is checked relative to some subset of previously proved lemmas. In fact, such a check is too difficult if there is no enough previously proved lemmas that support it. In this case, a skilled human verifier provides additional supporting lemmas based on his deep understanding of the DUV, which requires from the verifier a lot of knowledge, skills, and practice.

In contrast to theorem proving, equivalence checking and model checking can be fully automated and do not require human intervention. In the case of equivalence checking the question is whether two descriptions of the design are equivalent, it checks whether a DUV and a proved correct implementation for its specification have the same (or equivalent) behavior. The two compared designs are modeled using the same description method and combined in one design, called *miter*. A miter is constructed by adding the implementation of an equivalence function to perform a pairwise comparison between the corresponding outputs of the two given designs. The equivalence function takes outputs of compared designs as inputs and produces a single output which is the output of the miter. The output of the miter will be one iff at least one pair of outputs differ, otherwise it is proved to be constant zero and consequently the compared designs are proved to be equivalent—they produce identical output values under any input assignment. Automated detection for internal functional equivalences [65, 72] is utilized to decompose this verification proof into a set of simple problems. This makes equivalence checking applicable only if the compared designs have a fair degree of structural and functional similarity such that enough internal equivalences can be detected.

Model checking considers the question of how to verify that a given sequential design satisfies a temporal property. The DUV is modeled using a *Finite State Machine* (FSM). Each state of the FSM is identified with a Boolean assignment to a set of state variables. The specification is formulated as a set of properties, each property is a formula in a temporal logic such as *Computation Tree Logic* (CTL) [20] or *Linear Time Logic* (LTL) [80]. Approaches based on (incomplete) bounded model checking (BMC) [6] are efficient in finding counter examples that falsify a given property for a bounded number of states, however, it cannot prove that a temporal property is satisfied in all reachable states. Since 2003, model checking based on interpolation [69] has shown its advantages and is currently considered one of the most valuable complete formal verification methods. It derives an over-approximation for the forward image of the initial state of the FSM and iterates an *image operation* to compute an over-approximation for all reachable states. Because of the approximation, it is computationally feasible to prove that a given property is true for unbounded number of states. Nowadays, interpolation based model checking is complemented with IC3 [9, 10] which is currently the most powerful algorithm for model checking of hardware. IC3 maintains the sequence of stepwise over-approximating sets, because of approximation, some states are detected that lead to violating the given property although they are unreachable from the initial states. IC3 works by iteratively learning lemmas that demonstrate why these states cannot be reached within

a bounded number of steps. This automated algorithm of building inductive verification proofs allows a more powerful model checking for safety properties.

Decomposed problems by verification techniques are described by formulas of first order theories such as propositional logic, bit vectors, and linear arithmetic. For every decidable theory, there is a decision procedure that terminates with a correct answer to a decision problem for given formulas of this theory. *Boolean Satisfiability* (SAT) solvers are decision procedures for propositional logic that are utilized intensively by formal techniques. Given a Boolean formula, a SAT solver [37] decides whether the formula is satisfiable—there exists an assignment of its variables under which the formula evaluates to true; if it is satisfiable, it also reports the satisfying assignment. The success of SAT solvers can be largely attributed to their ability to learn from wrong assignments, prune large search spaces quickly and focus first on important variables—those variables that once given the right value, the problem is simplified significantly. *Satisfiability Modulo Theories* (SMT) [34, 73] is a generalization of SAT, it determines the satisfiability of a Boolean combination of formulas in more theories of first order logic via tailored decision procedures, e.g., SMT solver can decide the satisfiability of the data-path operations of a microprocessor at the word rather than the bit level.

Another proof solver that is utilized intensively besides SAT/SMT solvers is based on *Reduced Ordered Binary Decision Diagrams* (ROBDDs, or BDDs for short) [16]. BDDs are a highly useful graph-based data structure for manipulating Boolean formulas. The canonicity of this data representation is its main feature that enables a trivial tautology procedure for answering decision problems. If two formulas are functionally equivalent, then their BDD representations are isomorphic. One implication of canonicity is that all formulas that evaluate to true have the same BDD (a single node with a label '1') and all unsatisfiable formulas also have the same BDD (a single node with a label '0'). Thus, two formulas of completely different size can both be unsatisfiable, but their BDD representations are identical—a single node with the label '0'. As a consequence, checking for satisfiability can be done in constant time for a given BDD using this tautology procedure. However, building the BDD for a given formula can take exponential space and time, even if in the end it results in a single node. For formulas of some important functions, such as multiplication, constructing the BDD is infeasible due to its exponential size. For such arithmetic functions, word level decision diagrams such as *Multiplicative Binary Moment Diagram* (*BMD) [18] have been introduced, whereas they offer canonical representations for large multipliers. Unfortunately, the sizes of *BMDs are exponential for functions that can easily be represented using BDDs. For this

purpose, *Hybrid Decision Diagrams* such as HDDs [24] and K*BMDs [30] have been proposed. HDDs and K*BMDs are supersets of *BMDs and allow in all cases efficient and canonical representations as BDDs, in particular, for decision problems where data-path and control logic functions are combined. However, building word level decision diagrams for large circuits at the gate-level is computationally hard since the sizes of the diagrams may increase exponentially during the construction.

THESIS CONTRIBUTIONS

Verification of *Floating-Point Units* (FPUs) is usually among the top checklist items of any microprocessor. FPUs are data-intensive designs, achieving complete coverage on these is impossible through dynamic simulations. For example, the verification of a single operation with two 64-bit operands requires 2^{128} input data combinations, many life-years have to be spent to completely validate this operation. Formal verification is the only way to get a complete coverage on such designs. One noteworthy challenge is developing a fully automated technique which proves that a floating-point design is in consistence with the IEEE standard for floating-point arithmetic (IEEE Std 754-2008) [50]. Theorem provers have been applied extensively to verify properties of floating-point designs. Although a lot of automation has been added and floating-point libraries have been created to avoid repetition of proofs, the theorem proving methodology still requires an enormous amount of manual effort [90]. The paper by Jacobi et al. [54] is the highest automated work up to today, however, it skips the hardest part to verify—multiplication. In order to come up with a fully automated technique in this thesis, the functions of the fundamental elements of floating-point designs are first independently verified. Among these elements, the multiplier has turned out to be the toughest part to verify. Word level decision diagrams such as *BMDs suffer from an exponential blow-up of their size during the construction of the diagram from bit level formulas. Techniques based on SAT and SMT solvers fail to check for the correctness of large scale multiplier circuits in practical time. The most successful technique up to today is based on reverse engineering to an *Arithmetic Bit-Level* (ABL) representation of the circuit [96]. It extracts adder structures from gate-level netlists and builds full adder networks, however, building these adder networks is not possible for all multiplier architectures as well as for incorrect multipliers.

This dissertation investigates the problems of two distinctive formal verification techniques for verifying large scale multiplier circuits and proposes

two approaches to overcome some of these problems. The first technique is *equivalence checking based on recurrence relations* [85], while the second one is the *symbolic computation* technique which is based on the theory of *Gröbner bases* [87]. This investigation demonstrates that approaches based on symbolic computation have better scalability and more robustness than state-of-the-art equivalence checking techniques for verification of arithmetic circuits. According to this conclusion, the thesis leverages the symbolic computation technique to verify floating-point designs. It proposes a new algebraic equivalence checking [86], in contrast to classical combinational equivalence checking [66, 72], the proposed technique is capable of checking the equivalence of two circuits which have different architectures of arithmetic units as well as control logic parts, e.g., floating-point multipliers.

In the following, brief overviews are given about the three techniques that have been proposed to verify in a fully automated manner large scale multipliers as well as floating-point multipliers.

Recurrence Relations: Scalable Verification of Multipliers

State-of-the-art equivalence checking techniques [65] cannot deal with implementations that have few internal equivalences. This problem occurs especially for arithmetic circuits where one function can be implemented in many different ways [96]. For this reason, arithmetic properties of the multiplier function have been employed by equivalence checking based on recurrence relations in order to build a miter with many internal equivalences. The first approach in this context has been proposed by Fujita [42]. It is based on the fact that any function satisfying the recurrence relation $(X + 1) \cdot Y = X \cdot Y + Y$ is a multiplication. However, the original approach by Fujita does not scale, and it cannot verify multipliers larger than 16 bits.

To increase the scalability of this technique, we propose the *Checking Partial Product* (CPP) approach [85] which decomposes the verification process of the multiplier into a series of simpler cases. In every case, the generation and the addition of one partial product of the multiplier under verification will be checked using a recurrence equation. As shown in Figure 1, a given circuit netlist is decomposed into small parts depending on deduced information about the partial products of the circuit that is assumed to be a multiplier. The second step as shown in the figure is the construction of a miter for every decomposed part based on a recurrence relation. Finally, all created miters are checked independently by the *Combinational Equivalence Checking* (CEC) approach [72], as shown in the last block of Figure 1. The small differences and the similar ways

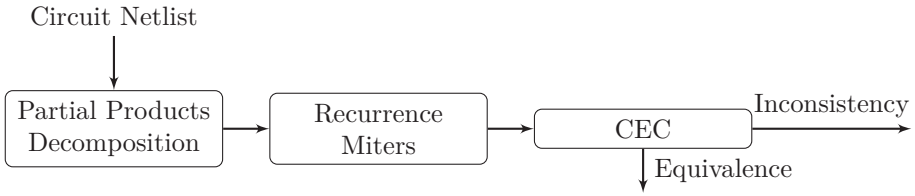

Figure 1: Checking Partial Product Approach

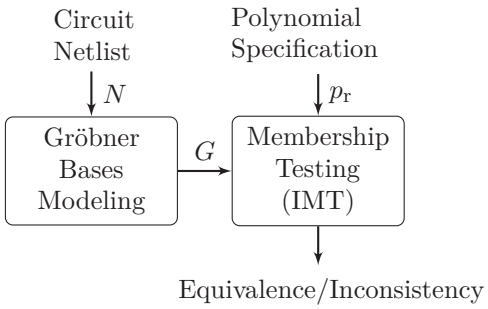

Figure 2: Flow of Symbolic Computation Technique

of carry propagation within miters allow fast equivalence checking, regardless of the multiplier size. As the multiplier size increases, the number of cases will increase, while the complexity to check one case will remain almost the same. Our approach is able to verify a multiplier at the gate-level without any information about its high-level specification or the internal structure of the netlist. The experiments show the capability of the proposed approach to verify up to 128-bit multiplier. However, the approach is not applicable for the verification of *Booth recoding* multipliers and optimized multipliers. The case splitting scheme of the approach assumes that the partial products are independent of each other, which is not the case for Booth recoding multipliers and optimized multipliers.

Symbolic Computation for Verifying Complex Multipliers

The symbolic computation technique [23, 39, 68, 77, 99, 100] reduces the verification problem to a membership testing of a specification polynomial in a set of multivariate polynomials modeling a circuit netlist. It solves the verification problem using an algebraic decision procedure called *Ideal Membership Testing* (IMT). As illustrated in Figure 2, the inputs of the IMT are the specification polynomial p_r of the circuit function and a set of polynomials in the form of Gröbner basis $G = \{g_1, \ldots, g_s\}$ modeling the circuit netlist N. The IMT proce-

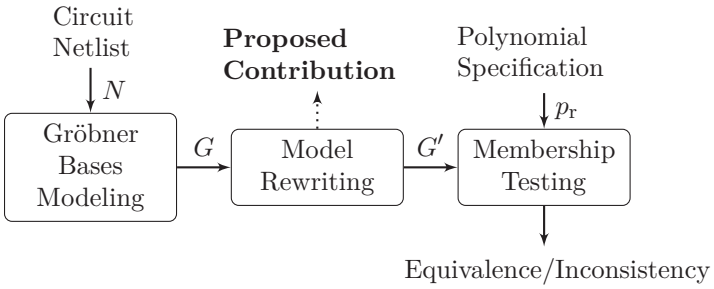

Figure 3: Symbolic Computation for Multipliers

dure answers the question of whether the circuit netlist satisfies its specification by applying *recursive divisions* on p_r wrt. G, denoted $p_r \xrightarrow{G}_+ r$, where r is the remainder of dividing p_r by G. The division steps are repeated until no term in r is divisible by the leading term of any polynomial in G. If $r = 0$, the circuit satisfies the specification, and an equivalence is proved, otherwise, an inconsistency between the model G and the specification is announced.

In the case of integer arithmetic, the IMT procedure suffers from an exponential increase in the size of the intermediate polynomial during the division (reduction) process, because of nonlinear terms that model carry chains which are propagated within bit-level implementations of integer multipliers. To improve the scalability of the technique, a rewriting step is inserted, as shown in Figure 3, to derive a new set of Gröbner basis polynomials $G' = \{g_1, \ldots, g_s\}$ from the algebraic model of the circuit $G = \{g_1, \ldots, g_t\}$, making the verification of a limited class of integer multipliers feasible. Model rewriting allows the early cancellation of shared terms in the polynomial representation, which effectively circumvents the blow-up within the IMT procedure. However, enhancing the technique by only rewriting is not sufficient to verify multipliers using complex architectures such as *Parallel Prefix Adders* (PPAs) or Booth recoding. The main reason—as identified by us—is the accumulation of *vanishing monomials*, which refers to monomials that always evaluate to zero.

We propose an algebraic algorithm which enables the verification of a large class of multiplier circuits, i.e., including basic and parallel multiplier architectures. Based on the observation of accumulating vanishing monomials, a novel rewriting scheme is proposed to reveal these monomials. In particular, the algorithm makes use of structural knowledge on the circuit netlist in order to identify and remove vanishing monomials early before starting the IMT process. Thus the approach can verify complex multiplier circuits of up to 128 bits in practical time.

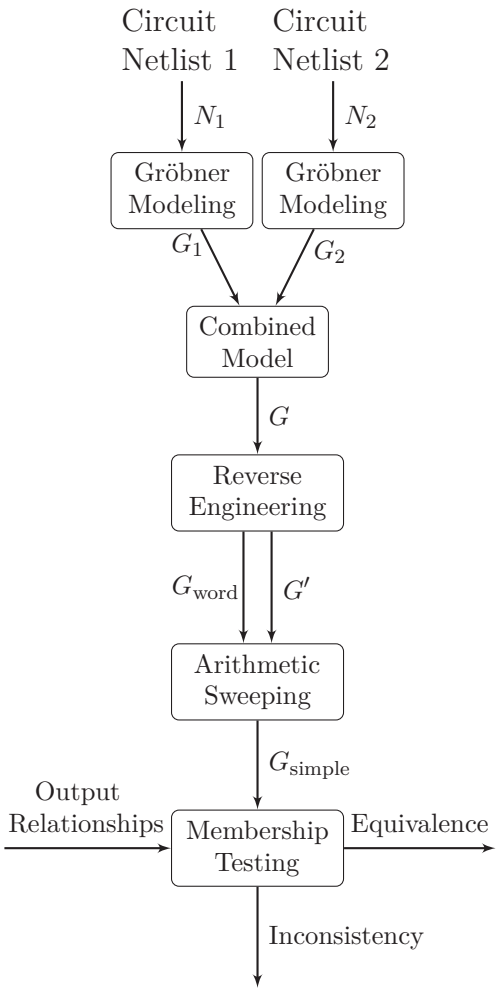

Figure 4: Abstracted Flow of ACEC

Equivalence Checking of Floating-Point Multipliers Using Gröbner Bases

Motivated by the fundamental problem that not every circuit specification p_{r} can be represented by one polynomial in a canonical and an abstract form over $\mathbb{Z}_{2^n}$, we are interested in equivalence checking, i.e., we want to prove the functional equivalence of two circuits. This can be done as follows: Assume the two circuits checked for equivalence represent the functions $f_1(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$ and $f_2(x_1, \ldots, x_n) = (z_1, \ldots, z_m)$ and are given as two sets of polynomials G_1 and G_2. Then we test the membership of each polynomial $z_j - y_j$ $(1 \leq j \leq m)$—which formulates the equivalence of each output bit—in polynomials from the combined model $G = G_1 \cup G_2$. This naïve method does

not scale since during the recursive reduction (division) process performed by the IMT procedure, the internal variables in the polynomials set G cause for a tremendous overhead which can only be resolved when the primary input variables x_i appear in the polynomials.

This problem can be circumvented if one knows internal equivalences in the two circuits, which allows putting internal variables into relation. Conceptually, this is similar to SAT sweeping [65] and as a consequence G is simplified. This ultimately avoids a blow-up of the polynomials during reduction. The difficulty is finding internal equivalences. To solve this problem we propose reverse engineering techniques: First, expected arithmetic word-level components such as multipliers and adders are detected in the circuit using structural signatures. Then, the proposed arithmetic sweeping uses the I/O boundaries of detected word-level components to prove internal equivalences and circumvent division blow-ups. To further reduce verification runtime during the divisions we propose a decomposition algorithm that allows more compact and semi-canonical representations for different implementations of the same function.

The result is a new *Algebraic Combinational Equivalence Checking* (ACEC) technique shown in Figure 4 which is based on Gröbner bases. It combines two Gröbner basis sets G_1 and G_2—modeling the compared netlists N_1 and N_2—in one Gröbner basis model G. Then it applies the two main algorithms of the ACEC, which are reverse engineering and arithmetic sweeping, as illustrated in Figure 4. The reverse engineering algorithm rewrites the model G into a new Gröbner basis G' to identify arithmetic functions of G' and abstract them to word-level polynomials, building from them a word-level model G_{word}. Using the arithmetic sweeping algorithm, polynomials of G_{word} and G' are leveraged to deduce and prove equivalence relationships between internal variables of G', which leads to a simplified Gröbner basis G_{simple} by merging internal variables of G' that are proved to be equivalent. Finally, as shown in the end of Figure 4, the ACEC checks the satisfiability of the output relationships in the simplified model G_{simple} using the IMT procedure; if all output relationships are satisfied, then the equivalence between N_1 and N_2 is proved, otherwise, the nonequivalence is announced. In contrast to classical combinational equivalence checking [66, 72], the ACEC can check the equivalence of two circuits which contain different architectures of arithmetic units, e.g., multipliers and adders, as well as control logic parts. Our experimental evaluation demonstrates the applicability of our algebraic equivalence checking approach on several optimized floating-point multipliers which cannot be verified by other fully automated proof techniques.

Outline

The dissertation is based on several peer-reviewed publications. The publications are listed individually for each of the three main chapters:

Chapter 2 provides the required background to keep this dissertation self-contained, it reviews definitions and notations from Boolean reasoning, formal verification, and symbolic computation technique.

Chapter 3 revisits equivalence checking based recurrence relations, it proposes a scalable verification approach for bit level multiplier circuits, the chapter is based on the publication:

Recurrence Relations Revisited: Scalable Verification of Bit Level Multiplier Circuits
Amr Sayed-Ahmed, Ulrich Kühne, Daniel Große, and Rolf Drechsler
IEEE Annual Symposium on VLSI (ISVLSI), 2015, 1–6.

Chapter 4 enhances the symbolic computation technique to verify complex architectures of integer multipliers, it is based on the publication which has been nominated for best paper at DATE 2016:

Formal Verification of Integer Multipliers by Combining Gröbner Bases with Logic Reduction
Amr Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler
Design, Automation and Test in Europe (DATE), 2016, 1048–1053.

Chapter 5 introduces a new algebraic equivalence checking technique for verifying formally floating-point circuits, the chapter is based on the following publication:

Equivalence Checking using Gröbner Bases
Amr Sayed-Ahmed, Daniel Große, Mathias Soeken, and Rolf Drechsler
Int'l Conf. on Formal Methods in CAD (FMCAD), 2016, 169–176.

Chapter 6 concludes the dissertation and provides directions for possible future work.

2

BACKGROUND

To keep this work self-contained, this chapter briefly provides the basics of the main state-of-the-art formal verification techniques for arithmetic circuits. Also, the theoretical background of the symbolic computation technique is introduced, which is the main focus of the thesis.

The chapter looks at the formal techniques leveraged for Boolean reasoning from two aspects: the first aspect is the modeling methodologies for the verification problem—the circuit under verification and its specification, the second aspect is the manipulation algorithms over the obtained models to perform the automated reasoning. So that in the first part of the chapter, various circuit modeling methods are provided. Then manipulation algorithms of three formal Boolean reasoning techniques are introduced, together with relating each technique to the respective modeling method.

The last part of the chapter reviews briefly theoretical backgrounds, capabilities, and drawbacks of formal verification techniques for arithmetic circuits, in addition to the specification of floating-point multiplier circuits as described in the IEEE standard for floating-point arithmetic [50] as well as various types of multiplier architectures.

2.1 CIRCUIT MODELING

Circuits are hardware implementations of Boolean functions, they are mainly classified as combinational or sequential circuits. While combinational circuits consist only of a combinational logic, sequential circuits integrate the combinational logic with memory elements such as flip-flops. Typically combinational logic is composed of the standard gates NOT, AND, OR, and XOR. A netlist of combinational logic gates can be thought of as a directed acyclic graph with wires carrying some value in the set $\{0, 1\}$, and these values are processed by gates computing specified Boolean functions. In the following, definitions of Boolean functions and different efficient modeling methods to manipulate the representation of the combinational logic of a circuit are given.

2.1.1 *Boolean Function*

$\mathbb{B} = \{0, 1\}$ is the set of Boolean values. A *Boolean variable* takes a value from this set.

Definition 1 (Boolean function)**.** *A multi-output Boolean function* $F : \mathbb{B}^n \to \mathbb{B}^m$ *is a Boolean function which maps* n *inputs to* m *outputs with* $n, m \in \mathbb{N}$*.*

The multiple-output function can be represented as a tuple $F = (f_1, \cdots, f_m)$ where $f_i : \mathbb{B}^n \to \mathbb{B}$ is an one-output function for each $i \in \{1, \cdots, m\}$. Hence $F(X) = (f_1(X), \cdots, f_m(X))$ over a finite set of Boolean variables X. The functions $f_i(X)$ are called *primary outputs* and the set of variables $X = \{x_1, x_2, \cdots, x_n\}$ are *primary inputs.*

Definition 2 (Integer-valued function)**.** *Some multi-output Boolean functions* $F : \mathbb{B}^n \to \mathbb{B}^m$ *such as arithmetic functions are encoded to integer-valued functions* $F : \mathbb{B}^n \to \mathbb{Z}_{2^m}$ *which map* n *Boolean inputs to an integer output* $\mathbb{Z}_{2^m}$*.*

Boolean functions of circuits are modeled by different representations such as *Truth-tables*, *Decision Diagrams* (DDs) [16], *And Inverter Graphs* (AIGs) [65], *Conjunctive Normal Forms* (CNFs) [98], or *Multivariate Polynomials* [68, 93]. The latter is based on *Boolean ring*—a concept from symbolic computation, while the others leverage *Boolean algebra* to represent efficiently Boolean functions.

Definition 3 (Boolean Algebra)**.** *A Boolean algebra is formally defined as a set of Boolean variables* $x_1, x_2, x_3, \cdots$*, three logic operations AND* $\wedge$*, OR* $\vee$*, and NOT* $\neg$*, and two distinct elements* 0 *and* 1 *such that the set holds the following properties:*

Idempotent:	$x_1 \wedge x_1 = x_1 \vee x_1 = x_1$
Complementation:	$x_1 \wedge \neg x_1 = 0$
	$x_1 \vee \neg x_1 = 1$
Identities:	$x_1 \wedge 1 = x_1 \vee 0 = x_1$
	$x_1 \wedge 0 = 0$
	$x_1 \vee 1 = 1$
Commutative:	$x_1 \wedge x_2 = x_2 \wedge x_1$
	$x_1 \vee x_2 = x_2 \vee x_1$
Associative:	$x_1 \wedge (x_2 \wedge x_3) = (x_1 \wedge x_2) \wedge x_3$
	$x_1 \vee (x_2 \vee x_3) = (x_1 \vee x_2) \vee x_3$
Absorption:	$x_1 \vee (x_1 \wedge x_2) = x_1 \wedge (x_1 \vee x_2) = x_1$

The Boolean algebra permits writing any Boolean function as *formulas*. For example, the following common functions could be described as follows:

XOR: $x_1 \oplus x_2 \equiv (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$

Implication: $x_1 \Rightarrow x_2 \equiv \neg x_1 \vee x_2$

Equivalence: $x_1 \leftrightarrow x_2 \equiv \neg(x_1 \oplus x_2)$

Definition 4 (Boolean Ring)**.** *A Boolean ring is a set on which the operations of multiplication $\cdot$, addition $+$, and subtraction $-$ over the Boolean elements of the set are defined and satisfy certain basic rules. The Boolean ring is given by $\mathbb{Z}$ modulo 2 denoted $\mathbb{Z}_2$, the elements of its set can take only two values $0, 1$ which are the Boolean values.*

For Boolean variables $x_1, x_2, x_3, \cdots$ in the set of the Boolean ring, the following rules are satisfied:

Additive inverse: $x_1 + -x_1 = 0$

Identities: $x_1 \cdot 1 = x_1 + 0 = x_1$

Commutative: $x_1 + x_2 = x_2 + x_1$

$x_1 \cdot x_2 = x_2 \cdot x_1$

Associative: $x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3$

$x_1 \cdot (x_2 \cdot x_3) = (x_1 \cdot x_2) \cdot x_3$

Distributive: $x_1 \cdot (x_2 + x_3) = x_1 \cdot x_2 + x_1 \cdot x_3$

Over the Boolean ring, Boolean functions can be represented as multivariate polynomials where the roots of polynomials are the truth assignments of the functions. This modeling method is explained in subsection 2.1.5.

2.1.2 *And Inverter Graph*

Definition 5 (And Inverter Graph (AIG))**.** *An AIG is a Boolean directed acyclic graph composed only of two-input AND gates and inverters (NOT gates).*

The AIG of a circuit is derived by factoring its gates into AND and OR gates, then converting OR gates into ANDs and inverters using *DeMorgan's rule*: $\neg(x_1 \vee x_2) \equiv (\neg x_1 \wedge \neg x_2)$.

Example 1. *Consider the full adder circuit shown in Figure 5. x_1, x_2 and x_3 are primary inputs, s and co are primary outputs, while v_1, v_2, v_3 and v_4 are*

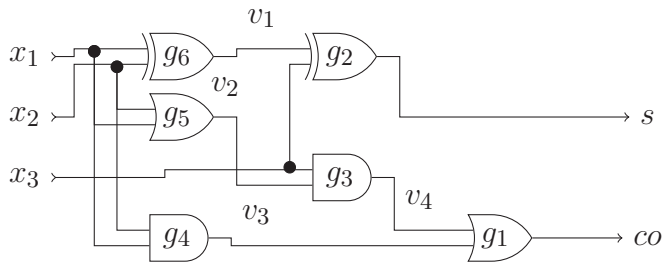

Figure 5: Full Adder Circuit

internal variables. The AIG model of the circuit is obtained by factoring the XOR gates and converting ORs as follows:

$$
\begin{array}{lll}
Gate: g_1 & co = v_4 \vee v_3 & co = \neg(\neg v_4 \wedge \neg v_3) \\
Gate: g_2 & s = v_1 \oplus x_3 & s = \neg(v_1 \wedge x_3) \wedge \neg(\neg v_1 \wedge \neg x_3) \\
Gate: g_3 & x_4 = v_2 \wedge x_3 & v_4 = v_2 \wedge x_3 \\
Gate: g_4 & v_3 = x_1 \wedge x_2 & v_3 = x_1 \wedge x_2 \\
Gate: g_5 & v_2 = x_1 \vee x_2 & v_2 = \neg(\neg x_1 \wedge \neg x_2) \\
Gate: g_6 & v_1 = x_1 \oplus x_2 & v_1 = \neg(x_1 \wedge x_2) \wedge \neg(\neg x_1 \wedge \neg x_2)
\end{array}
$$

AIGs consist of specific types of nodes: two-input AND nodes, primary input (PI) nodes, and primary output (PO) nodes. Primary input nodes have no incoming edges. The inverters are not counted as nodes of the AIG graph, they are represented as complemented edges. Figure 6 shows an AIG representation for the full adder of Example 1, the AND nodes are represented as circles, the complemented edges as dashed arrows, x_1, x_2, and x_3 are input nodes, while s and co are output nodes.

Different manipulation methods are applied on the AIG model to minimize the number of ANDs and inverters such as *structural hashing* and *rewriting* [71]. Structural hashing ensures during the construction of the AIG that no two AND gates have identical pairs of incoming edges, as is noticed in Figure 6. Rewriting selects iteratively AIG subgraphs rooted at a node and replaces them with smaller precomputed subgraphs, while preserving the functionality of the root node. The subgraphs are collapsed by refactoring their Boolean expressions [12] or balancing them using the algebraic tree-height reduction [25].

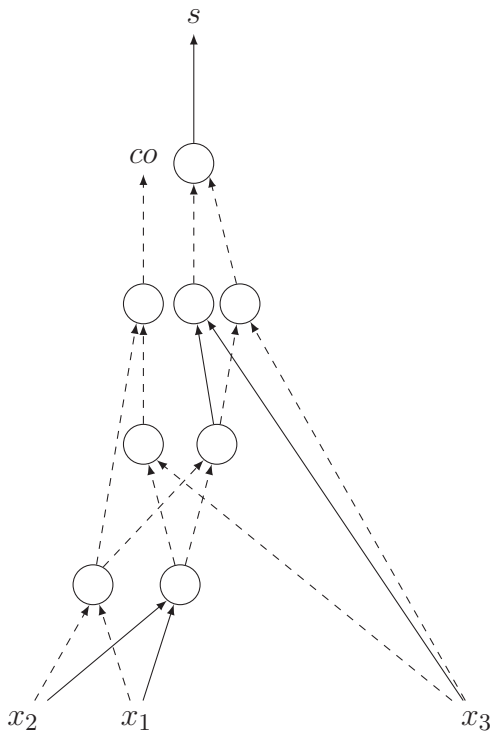

Figure 6: AIG for Full Adder

2.1.3 *Conjunctive Normal Form*

Definition 6 (Conjunctive Normal Form)**.** *A formula (one-output Boolean function) is in Conjunctive Normal Form* (CNF) *if it is a conjunction of disjunctions of literals, i.e., it has the form* $\bigwedge_i (\bigvee_j l_{ij})$ *where* l_{ij} *is the* j*-th literal (a literal is either a Boolean variable or its negation) in the* i*-th clause (a clause is a disjunction of literals).*

The transformation from circuit gates into CNFs is done via Tseitin's encoding [98]. Several clauses are added to constrain the value of the output variable

of each logic gate according to the function of the gate. For Boolean functions of basic gates, the clauses modeling these functions are:

$$
\begin{aligned}
x_o = \neg x_1 \quad &\Longrightarrow (x_o \vee x_1) \ \wedge \ (\neg x_o \vee \neg x_1) \\
x_o = x_1 \wedge x_2 \quad &\Longrightarrow (\neg x_o \vee x_1) \ \wedge \ (\neg x_o \vee x_2) \ \wedge \ (x_o \vee \neg x_1 \vee \neg x_2) \\
x_o = x_1 \vee x_2 \quad &\Longrightarrow (x_o \vee \neg x_1) \ \wedge \ (x_o \vee \neg x_2) \ \wedge \ (\neg x_o \vee x_1 \vee x_2) \\
x_o = x_1 \oplus x_2 \quad &\Longrightarrow (\neg x_o \vee x_1 \vee x_2) \ \wedge \ (x_o \vee \neg x_1 \vee x_2) \ \wedge \ (x_o \vee x_1 \vee \neg x_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \ (\neg x_o \vee \neg x_1 \vee \neg x_2).
\end{aligned}
$$

CNF clauses for a complete circuit consist of the conjunction of all CNFs of the local gates. The transformation into CNF clauses increases the size of the Boolean function linearly, however, particular decision procedures—SAT solvers—are designed to work efficiently over CNF models, see Subsection 2.2.1.

2.1.4 *Decision Diagrams*

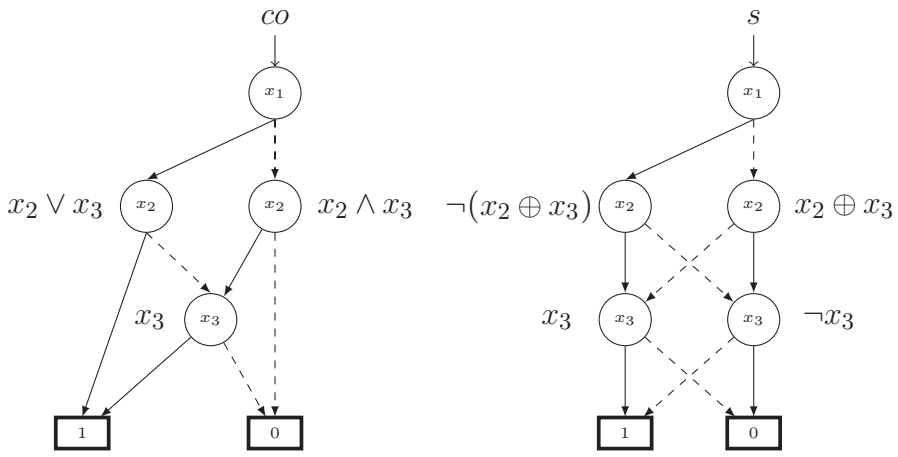

Figure 7: BDD for Full Adder

Decision Diagrams (DDs) are directed acyclic graphs that are built by ordering primary inputs of a function and applying recursive decompositions to the function based on this order. DDs such as *Binary Decision Diagrams* (BDDs) [15, 16] and *Multiplicative Binary Moment Diagrams* (*BMDs) [17] are popular data structures since they offer canonical representations for Boolean functions and efficient manipulation methods, making checking of functional properties such as satisfiability and equivalence straightforward. BDDs map Boolean functions $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ based on *Shannon's decomposition* rule

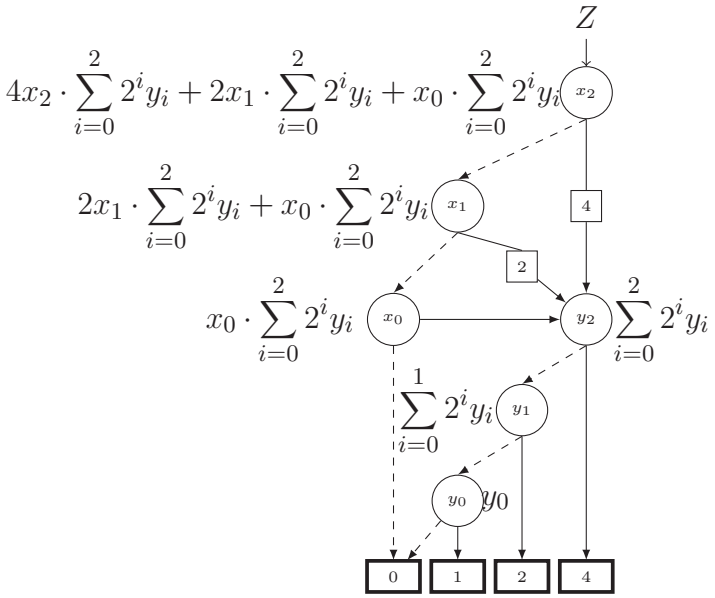

Figure 8: *BMD for 3-bit Multiplier

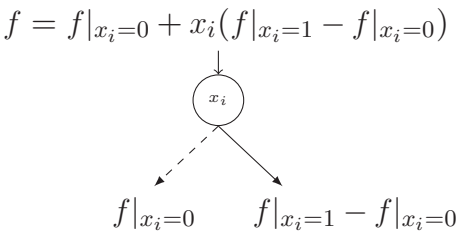

Figure 9: Decomposition Rule of *BMD

$f = \neg x_i f|_{x_i=0} \vee x_i f|_{x_i=1}$. Term $f|_{x_i=1}$ is the positive cofactor of f with respect to the variable x_i, i.e., the function that is resulted when the value one is assigned to x_i. In a similar way, $f|_{x_i=0}$ denotes the negative cofactor of f, when $x_i = 0$. For *BMDs, *Positive Davio's* rule $f = f|_{x_i=0} + x_i(f|_{x_i=1} - f|_{x_i=0})$ is the basis to represent a Boolean function at word level as an integer-valued function $f : \mathbb{B}^n \to \mathbb{Z}_{2^m}$. The Positive Davio's rule decomposes f into the negative cofactor $f|_{x_i=0}$ and the function $(f|_{x_i=1} - f|_{x_i=0})$ which is called the linear moment of f.

The graphs of DDs as shown in Figure 7 and Figure 8 consist of non-terminal vertices (drawn as circles) and terminal vertices (drawn as squares) labeled by the possible values of the represented function. Each non-terminal vertex is labeled by a variable from the primary inputs of the function and has exactly two children. The directed edges to these children are called low-edge and high-edge

and are drawn dashed and solid, respectively. A non-terminal vertex labeled x_i represents a function f decomposed wrt. x_i using a rule into two child functions which are represented by children vertices. In the case of BDD, the child functions are the negative and positive cofactors of f, while for *BMD, as shown in Figure 9, f is decomposed into negative cofactors and linear moment functions.

Example 2. *Consider the BDD representations of a full adder shown in Figure 7. x_1, x_2 and x_3 are primary inputs, while $s = x_1 \oplus x_2 \oplus x_3$ and $co = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ are primary outputs.*
The variables are ordered as $x_1 > x_2 > x_3$, based on this chosen order, the function of the output co is decomposed first wrt. x_1 by Shannon's rule into $co = \neg x_1 co|_{x_1=0} \vee x_1 co|_{x_1=1}$, whereas $co|_{x_1=0} = x_2 \wedge x_3$ and $co|_{x_1=1} = x_2 \vee x_3$ are the resulted cofactors. This decomposition is represented in Figure 7 by a vertex labeled x_1 with two children vertices that represent the child functions. The two children vertices are labeled by the variable x_2 since it is chosen to decompose the negative cofactor $co|_{x_1=0} = \neg x_2 co|_{x_1=0\&x_2=0} \vee x_2 co|_{x_1=0\&x_2=1}$ into $co|_{x_1=0\&x_2=0} = 0$ and $co|_{x_1=0\&x_2=1} = x_3$ and to decompose the positive cofactor $co|_{x_1=1}$ into $co|_{x_1=1\&x_2=0} = x_3$ and $co|_{x_1=1\&x_2=1} = 1$. Finally x_3 decomposes the resulted child functions from the two previous steps into 0 and 1. A similar way is applied to build a BDD for the output function s, which is shown in the right side of Figure 7.

*BMDs support multiplicative edge weights—the values at the edges are multiplied with the represented child functions. Using this data structure, it is feasible to build compact diagrams for multipliers of large bit width, whereas multiplier representations based on BDDs have exponential sizes.

Example 3. *As shown in Figure 8, *BMD can represent a 3-bit multiplier $Z = \sum_{i=0}^{2} 2^i x_i \cdot \sum_{i=0}^{2} 2^i y_i$ by a compact diagram. The chosen order is $x_2 > x_1 > x_0 > y_2 > y_1 > y_0$.*
The function $Z = 4x_2 \cdot \sum_{i=0}^{2} 2^i y_i + 2x_1 \cdot \sum_{i=0}^{2} 2^i y_i + x_0 \cdot \sum_{i=0}^{2} 2^i y_i$ is decomposed wrt. x_2 by Positive Davio's rule as $Z = Z|_{x_2=0} + x_2(Z|_{x_2=1} - Z|_{x_2=0})$. The negative cofactor function is $Z|_{x_2=0} = 2x_1 \cdot \sum_{i=0}^{2} 2^i y_i + x_0 \cdot \sum_{i=0}^{2} 2^i y_i$, while the positive-edge is labeled by '4' to represent the coefficient of the linear moment function $Z|_{x_2=1} - Z|_{x_2=0} = 4 \sum_{i=0}^{2} 2^i y_i$. Decomposing these child functions recursively by the same explained way builds the diagram shown in Figure 8.

Because some functions can only be represented efficiently by *BMDs and others can easily be represented by BDDs [30], hybrid DDs such as HDDs [24] and K*BMDs [30] have been proposed to model more diversified types of functions by one diagram. They support mixing different decomposition types—the function decompositions are not performed by one decomposition rule as in BDD and *BMD. Every variable is decomposing the function using one of three decomposition rules: Shannon, Positive Davio, or *Negative Davio* $f = f|_{x_i=1} + (1 - x_i)(f|_{x_i=0} - f|_{x_i=1})$. Choosing a proper variable ordering and relating every variable with the appropriate decomposition type are the key roles for efficient modeling of a given function by hybrid DDs, however, finding such good choices as well as restricting the weights of edges to make the graph canonical are difficulties that restrict applications of hybrid DDs.

2.1.5 *Multivariate Polynomials*

Symbolic computation provides the Gröbner bases theory which is capable of modeling a circuit as a set of Boolean polynomials. This subsection gives an overview about the practical usage of the theory for the circuit modeling, skipping over the theoretical part which is presented in Subsection 2.2.3.

Definition 7. *A* Boolean polynomial $p = c_1M_1 + \cdots + c_tM_t$ *is a finite sum of terms, where each term is the product of a coefficient* c_i *and a power product over a set of* n *Boolean variables* $\{x_1, \ldots, x_n\}$ *denoted a* monomial $M_i = x_1x_2 \cdots x_{n-1}x_n$. *The coefficients are integers—*$c_i \in \mathbb{Z}$ *for all* $i \neq 1$, *only the leading coefficient* $\mathrm{lc}(p) = c_1 \in \{-1, 1\}$ *is limited to '*-1*' or '*1*'*

The monomials of a polynomial are ordered according to a *monomial ordering* $\prec$, such that $M_1 > \cdots > M_t$, the *leading term* of the polynomial is $\mathrm{lt}(p) = c_1M_1$, the *leading monomial* is $\mathrm{lm}(p) = M_1$, and the *leading coefficient* is $\mathrm{lc}(p) = c_1$. The thesis denotes $\mathrm{tail}(p) = p - \mathrm{lt}(p) = c_2M_2 + \cdots + c_tM_t$.

A set of Boolean polynomials $P = \{p_1, \ldots, p_s\}$ belongs to a *Boolean Polynomial Ring* $\mathbb{Z}_2(x_1, \ldots, x_n)$, where $(\mathbb{Z}_2)$ is the Boolean ring (see Definition 4). Within the polynomial ring $\mathbb{Z}_2(x_1, \ldots, x_n)$, the set of polynomials $\langle x_i^2 - x_i \rangle$ are added to keep the variables x_i in the Boolean domain. Note that the solutions of the polynomial equation $x_i^2 - x_i = 0$ are $x_i \in \{0, 1\}$, which restrict values of x_i to Boolean values. The practice influence of these polynomials $\langle x_i^2 - x_i \rangle$ is interpreted by reducing $x_i^{\alpha_i}$ to x_i every time its degree becomes greater than one during any computational step. For example, the monomial $x_1^2x_2^3x_3$ is equal to $x_1x_2x_3$ over the Boolean polynomial ring.

For modeling circuits, the monomial order follows the reverse topological order of the variables of the modeled circuit. Logic gates of a circuit are modeled by polynomials and signals as Boolean variables. The modeling is performed according to the Boolean function of gates, for the basic Boolean functions, the polynomial representations are as follows:

$$\begin{array}{lll}
\text{NOT:} & x_o = \neg x_1 & \Longrightarrow\ -x_o - x_1 + 1 \\
\text{AND:} & x_o = x_1 \wedge x_2 & \Longrightarrow\ -x_o + x_1x_2 \\
\text{OR:} & x_o = x_1 \vee x_2 & \Longrightarrow\ -x_o - x_1x_2 + x_1 + x_2 \\
\text{XOR:} & x_o = x_1 \oplus x_2 & \Longrightarrow\ -x_o - 2x_1x_2 + x_1 + x_2 \\
\text{MUX:} & x_o = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3) & \Longrightarrow\ -x_o + x_1x_2 - x_1x_3 + x_3.
\end{array}$$

Each Boolean function is modeled in a way that the output variable x_o is described in terms of the input variables $\{x_1, x_2, x_3\}$. The solutions of the polynomial equations of these Boolean functions correspond to the truth assignments of these functions.

Example 4. *For the NOT function the solutions (roots) of its polynomial are the pairs* $(x_o = 0,\ x_1 = 1)$ *and* $(x_o = 1,\ x_1 = 0)$ *which are the truth assignment of the function.*

By ordering each variable of the model according to its reverse topological level in the circuit, every polynomial will be of the form $p_i := x_i + \text{tail}(p_i)$, where x_i is the gate's output variable and $\text{tail}(p_i)$ are terms consisting of the gate's input variables, describing the function implemented by the gate. According to this polynomial form, all leading monomials of the model will be relatively prime, which is the main condition to represent a set of polynomials as Gröbner basis (see Subsection 2.2.3).

Example 5. *Consider the full adder circuit implementing the function* $s + 2co = x_1 + x_2 + x_3$ *shown in Figure 5. Its algebraic model is:*

$$\begin{array}{ll}
g_1 := -co - v_4v_3 + v_4 + v_3 & g_2 := -s - 2v_1x_3 + v_1 + x_3 \\
g_3 := -v_4 + v_2x_3 & g_4 := -v_3 + x_1x_2 \\
g_5 := -v_2 - x_1x_2 + x_1 + x_2 & g_6 := -v_1 - 2x_1x_2 + x_1 + x_2.
\end{array}$$

Ordering the polynomial variables in the reverse topological order of the circuit yields $co > s > v_4 > v_3 > v_2 > v_1 > x_3 > x_2 > x_1$. *Following this order, the leading monomials of all polynomials will be relatively prime, e.g., the leading monomial of* g_1 *is* co, *and it is prime relative to all other leading monomials, the extracted algebraic model is therefore a Gröbner basis.*

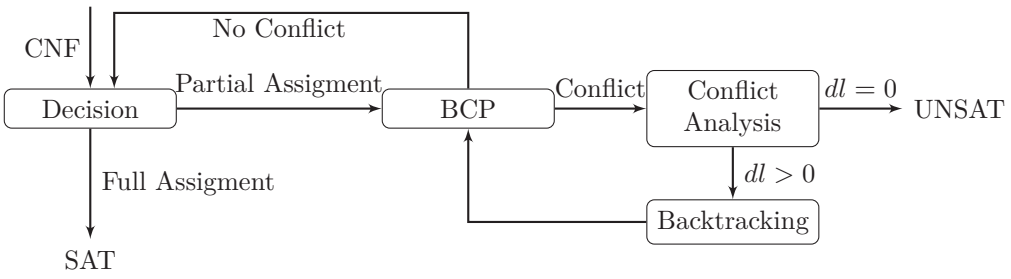

Figure 10: DPLL Algorithm

The reader can find more mathematical details about the Gröbner bases theory and the symbolic computation in Subsection 2.2.3.

2.2 BOOLEAN REASONING

For formal verification of circuits, there are mainly three proof techniques utilized for efficient Boolean reasoning of problems derived from circuits. Traditionally, Satisfiability (SAT) and Decision Diagrams (DDs)—in particular BDDs—are used intensively to solve different formal verification problems, however, they are incapable of dealing with problems that require to solve nonlinear arithmetic constraints which can be solved by the symbolic computation technique as is demonstrated in the thesis. In this section, the main concepts of SAT and BDDs are given, then concepts and notations of the symbolic computation technique utilized in the thesis are described.

2.2.1 *Boolean Satisfiability*

The Boolean satisfiability problem is about finding an assignment that satisfies a set of constraints. Because of that, it has a practical and theoretical importance in many applications which has led to a vast amount of research to develop powerful SAT solvers.

Definition 8 (Assignment)**.** *Given a Boolean function $f(x_1, \cdots, x_n)$, an assignment $\alpha = (a_1, \cdots, a_n)$ to f is mapping each primary inputs x_i to elements of Boolean values $a_i \in \{0, 1\}$. The assignment is full if all primary inputs are assigned, and partial otherwise.*

Definition 9 (Boolean Satisfiability Problem (SAT))**.** *Given an one-output Boolean function (formula)* $f(x_1, \cdots, x_n) : \mathbb{B}^n \rightarrow \mathbb{B}$*, the Boolean satisfiability problem decides whether there exists an assignment* $\alpha = (a_1, \cdots, a_n)$ *for primary inputs of* f *under which the formula evaluates to true (*$f(\alpha) = 1$*), otherwise,* f *is proved unsatisfiable since there is no such an assignment.*

Given a Boolean formula f, a SAT solver decides whether f is satisfiable and reports a satisfying assignment, or it proves that f is unsatisfiable. Typically, SAT solvers consider the problem of solving formulas in CNF (see Subsection 2.1.3) since every formula can be converted to this form in a linear time and the *Davis-Putnam-Loveland-Logemann* (DPLL) algorithm [28] is performed efficiently over it. To find a satisfiable solution, the majority of SAT solvers leverage the DPLL algorithm to perform the search progress by making a decision about a value of a Boolean variable, propagating implications of this decision, and backtracking in the case of a conflict. This search progress can be thought of as traversing and backtracking on the binary tree of the search space. In this tree, internal nodes represent partial assignments and the leaves represent full assignments. Each decision is associated with a *decision level* which is the depth in the binary decision tree and denotes the number of variables assigned by previous decisions.

The DPLL algorithm performs its steps based on the status of CNF clauses under assignments. A clause is *satisfied* if one or more of its literals are satisfied, *conflicting* if all of its literals are assigned but not satisfied, *unit* if it is not satisfied and all but one of its literals are assigned, and *unresolved* otherwise. The DPLL algorithm as represented by [105] consists of main four steps, as shown in Figure 10:

1. *Decision*, it chooses an unassigned variable and assigns value for it. If and only if there are no more variables to assign, the solver announces the problem as satisfiable and reports a satisfiable assignment. There are numerous heuristics for making these decisions which are associated with decision levels.

2. *Boolean constraints propagation* (BCP), based on the decision taken by the previous step, the *unit clause rule* is applied repeatedly until either a conflict is encountered or there are no more implications. The rule is applied on a clause at the unit status—with a single unassigned literal—by assigning to the unassigned literal the value that evaluates the clause to true.

3. *Conflict Analysis*, it returns a decision level which is utilized by the solver for backtracking. If a conflict at decision level 0 is detected, the solver proves that the formula is unsatisfiable, otherwise, it backtracks to the decision level given by the conflict analysis. In addition to computing the backtracking level and detecting the problem unsatisfiability, this analysis step adds new constraints to the search in the form of new clauses to avoid the future occurrence of this conflict, this approach is named *conflict-based-learning*.

4. *Backtracking*, based on the decision level (dl) generated by the conflict analysis, it erases all variable assignments at decision levels larger than dl.

For more details about SAT/SMT solvers, the interested reader is referred to the book [64].

2.2.2 *Binary Decision Diagrams*

The canonicity of BDDs allows to solve decision problems such as satisfiability or unsatisfiability in a constant time for given *reduced* BDDs since two functions are identical iff their reduced BDDs are identical. A BDD is a canonical representation under two conditions:

1. The BDD is reduced by reductions rules until neither of the rules is applicable.

2. The variables of the reduced BDD appear in the same order $x_1 < x_2 < \cdots < x_n$ on each path from the root node to a terminal node (the BDD is ordered).

A BDD can be implemented as a simple binary decision tree where each variable appears at least once from the root to the leaves. Such a representation has the same size of the truth table of the function since every path in this BDD from the root to a leaf corresponds to an assignment in the truth table. The BDD provides the feature that it can be reduced from such a tree to a unique representation under specific reduction rules which are repeated as long as they can be applied.

1. Reduction #1 merges isomorphic subtrees, isomorphic subtrees are those that have roots representing the same variable and have the same left and right children.

2. Reduction #2 removes redundant nodes which their values do not affect the values of paths that go through it. At this case, the two edges of the redundant node point to the same child node, the redundant node is removed by redirecting its incoming edge to its child node.

The size of a BDD as well as its canonicity depend strongly on the variable ordering, whereas different orders construct different BDDs for a given function. To obtain canonical BDDs for different implementations of a given function, their reduced BDDs must have a fixed variable ordering. In addition to that, there are functions such as adders which their reduced BDDs have a polynomial number of nodes under a certain variable ordering, while with another order the size of the diagram is exponential. For finding a good variable ordering of a given function, efficient heuristic variable ordering algorithm has been proposed, based on both static and dynamic ordering [81].

However, building a canonical decision diagram for a given function may take exponential space and time, even if in the end it results in a bounded number of nodes. To avoid the blow-up during building a BDD, instead of creating the BDD directly for a given function, the diagram is composed recursively from BDDs of its subexpressions. The algorithm that performs such a recursive composition is known as the *Synthesis* (also called *Apply*) algorithm. Typically for this, efficient implementations use a recursive synthesis algorithm [8] based on the *if-then-else* operator (ITE). ITE is a Boolean function defined as $\text{ite}(f, g, h) = (f \wedge g) \vee (\neg f \wedge h)$, it can express main Boolean operators between BDDs as follows:

$$\neg f = \text{ite}(f, 0, 1) \qquad f \wedge g = \text{ite}(f, g, 0)$$
$$f \vee g = \text{ite}(f, 1, g) \qquad f \oplus g = \text{ite}(f, \neg g, g).$$

For BDDs of three functions f, g and h that at least two of them share the root nodes of variable x, a BDD of the operator $\text{ite}(f, g, h)$ is constructed by calling recursively the following function:

$$\text{ite}(f, g, h) = \text{ite}(x,\ \text{ite}(f|_{x=1}, g|_{x=1}, h|_{x=1}),\ \text{ite}(f|_{x=0}, g|_{x=0}, h|_{x=0})).$$

Starting with the top most variable x, this equivalence formula is applied recursively to all the variables in the order they appear in their respective BDDs (f, g, and h must have compatible ordering for the algorithm to work). The operation ITE increases the number of isomorphic subtrees, therefore, the sizes of BDDs are reduced efficiently by the reduction rule #1.

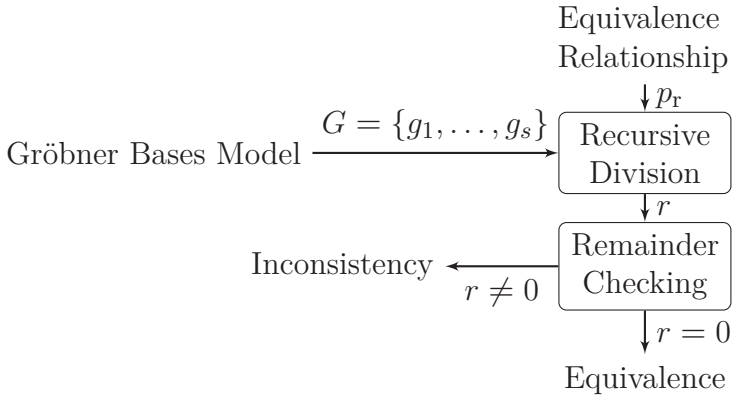

Figure 11: Ideal Membership Testing (IMT)

A tautology procedure based on BDDs solves decision problems by representing their functions as reduced BDDs under a fixed variable ordering. These problems can be classified mainly into two types:

1. The equivalence testing between reduced BDDs of two functions f and h. This equivalence test is very easy since it suffices to check whether the roots for f and h lead to the same node, which can be done in a constant time.

2. The satisfiability problem to find an assignment α for which $f(\alpha) = 1$. This is done by a simple depth-first search approach which is given the reduced BDD of f in order to find a path from the root of f to the 1-sink, the assigned values for the variables in such a path is the assignment α. Otherwise, if there is no such a path, then f is unsatisfiable.

The interested reader is referred to [32] for more details about decision diagrams.

2.2.3 *Symbolic Computation*

Symbolic computation offers an algebraic decision procedure named *Ideal Membership Testing* (IMT) which can answer questions about the correctness of equivalence relationships. As shown in Figure 11, the IMT takes two inputs: 1) the circuit model as a Gröbner basis $G = \{g_1, \ldots, g_s\}$ (see Subsection 2.1.5), and 2) a multivariate polynomial p_r describing the equivalence relationship between two or multiple variables of the circuit model. It tests whether the equivalence relationship polynomial p_r lies in the Gröbner basis $G = \{g_1, \ldots, g_{\hat{s}}\}$, the

testing is performed by reducing (dividing) p_{r} wrt. G. In case that the remainder r of applying the division algorithm to divide p_{r} by G is equal to zero, then IMT proves that the circuit satisfies the equivalence relationship, otherwise r is a symbolic polynomial constructed of the primary inputs of the circuit model and IMT announces that the relationship is not satisfied in the model.

Because the IMT is applied on Gröbner bases, it is essential to show that the modeling method introduced in Subsection 2.1.5 offers a Gröbner basis model for a given circuit. By modeling every logic gate in the circuit as one polynomial, a set of Boolean polynomials is constructed. This set $P = \{p_1, \ldots, p_s\} \in \mathbb{Z}_2(x_1, \ldots, x_n)$ has a set of all solutions named *affine variety* $V(p_1, \ldots, p_s)$ of the polynomial equations $p_1(x_1, \ldots, x_n) = \cdots = p_s(x_1, \ldots, x_n) = 0$. The affine variety is not only the solutions of the given set of polynomials P, in fact, it is the set of solutions of the ideal generated by the polynomials. An *ideal* $I = \langle P \rangle = \{\sum_{i=1}^{s} h_i \cdot p_i : h_i \in \mathbb{Z}_2(x_1, \ldots, x_n)\}$ is generated by this set of polynomials P, and we call P the bases (generators) of the ideal I. The ideal I may have many other bases. The bases are different representations of the set of polynomials P. One of these bases is called a Gröbner basis $G = \{g_1, \ldots, g_{\hat{s}}\}$, for which $V(G) = V(I)$.

Buchberger (1965) [19] introduced the algorithmic theory of Gröbner bases which are primarily defined for ideals in a polynomial ring $K[x_1, \cdots, x_n]$ over a *field* K [26]. To apply the Gröbner bases theory to polynomial rings over a *ring*, various approaches have been proposed [3, 57, 58, 95] to extend basic definitions and concepts. Recently, there is renewed interest to extend the theory for more types of rings [41, 44, 83].

Definition 10 (Ring)**.** *A ring is a set with two operations addition and multiplication satisfying additive and multiplicative associativity, additive commutativity, left and right distributivity, and existence of additive identity and inverse. A commutative ring also satisfies multiplicative commutativity.*

Definition 11 (Field)**.** *A field K is a commutative ring with unity, where every element in K, except 0, has a multiplicative inverse (i.e., $\forall x \in K - \{0\}$, $\exists \hat{x} \in K$ such that $x \cdot \hat{x} = 1$).*

The integers $\mathbb{Z}_{2^m}$ (for $m > 1$) is not a field since not every element in $\mathbb{Z}_{2^m}$ has an inverse, it is a commutative ring. For example, $\mathbb{Z}_4$ is not a field because it has the element 2 whose multiplicative inverse 0.5 is not an integer and is not in $\mathbb{Z}_4$.

As shown in Subsection 2.1.5, to model designs implementing integer-valued functions (see Definition 2) as multivariate polynomials, the variables of the

polynomials are Booleans while the coefficients are in the integer ring $\mathbb{Z}$, therefore, the ideals of those polynomials will be in the Boolean ring $\mathbb{Z}_2(x_1, \ldots, x_n) = \mathbb{Z}[x_1, \ldots, x_n]/\langle x_1^2 - x_1, \cdots, x_n^2 - x_n\rangle$, whereas the ideal $\langle x_i^2 - x_i\rangle$ restricts the values of x_i to the set $\{0, 1\}$. In [41], the ring $\mathbb{Z}[x_1, \ldots, x_n]/I_a$ has been investigated, where $I_a \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ is an ideal, it has proposed a restriction on I_a to arrive at a necessary and sufficient condition, such that all ideals in $\mathbb{Z}[x_1, \ldots, x_n]/I_a$ are isomorphic to integer lattices, therefore, Gröbner bases representations of those ideals can be derived as for fields. The restriction is that the Gröbner basis G_a of I_a consists of *monic* polynomials, one for every variable x_i.

Definition 12 (Monic Polynomial)**.** *A Polynomial is named monic, if it is in the form* $x^{\alpha_t} + c_{t-1}x^{\alpha_{t-1}} + \cdots + c_1x^{\alpha_1} + c_0$*, whereas* $\alpha_1 \in \mathbb{N}$ *and the leading coefficient of the leading term* x^{α_t} *is equal to one.*

Theorem 1 ([41])**.** *Let* $I_a \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ *be a non-zero ideal. Let* G_a *be a Gröbner basis for* I_a *wrt. a monomial ordering,* $\prec$*. If polynomials of* G_a *are monic then the Gröbner bases theory can be applied on ideals of the ring* $\mathbb{Z}[x_1, \ldots, x_n]/I_a$*.*

Since the ideal $\langle x_i^2 - x_i\rangle$ satisfies the condition of Theorem 1, Gröbner bases could be derived for ideals in the Boolean ring $\mathbb{Z}_2(x_1, \ldots, x_n)$.

To compute the Gröbner basis $G = \{g_1, ..., g_{\hat{s}}\}$ for an ideal $I\langle p_1, \ldots, p_s\rangle$, a *Gröbner bases algorithm* constructs G in a finite number of steps by applying *S-polynomial* $\mathrm{Spoly}(p, g) \xrightarrow{G}_+ r$ in every step. For fields, *Buchberger's algorithm* [19] is applied, while for rings such as $\mathbb{Z}_2(x_1, \ldots, x_n)$ the Gröbner bases algorithm proposed by [58] generates a Gröbner basis by computing S-polynomials for rings. The Gröbner basis is obtained by applying repeatedly S-polynomial until: 1) every p_i can be reduced to zero wrt. G; and 2) S-polynomial can reduce every pair of polynomials in G to zero.

Definition 13 (S-polynomial for Fields [26])**.** *The* S-polynomial *of polynomials* p *and* g *in a polynomial set* P*, is the combination* $\mathrm{Spoly}(p, g) = \frac{L}{\mathrm{lt}(p)} \cdot p - \frac{L}{\mathrm{lt}(g)} \cdot g$*, where* L *is the least common multiple* $\mathrm{LCM}(\mathrm{lm}(p), \mathrm{lm}(g))$*. Note that* $\mathrm{Spoly}(p, g)$ *cancels the leading terms of* p *and* g*, the remainder* r *obtained in* $\mathrm{Spoly}(p, g) \xrightarrow{P}_+ r$ *gives a new leading term.*

Definition 14 (S-polynomial for Rings [58])**.** *For rings, the S-polynomial is the combination* $\mathrm{Spoly}(p, g) = \frac{L}{\mathrm{lm}(p)} \cdot p - \frac{L}{\mathrm{lm}(g)} \cdot q_c \cdot g$*, whereas* $lc(p) = q_c \cdot lc(g) + r_c$*,* q_c *is the quotient of dividing* $lc(p)$ *by* $lc(g)$ *and* r_c *is the remainder.*

The utilized modeling method in the thesis models the circuit directly as a Gröbner basis and avoids the Gröbner bases algorithm which is computationally expensive, however, we are interested in the S-polynomial since it tests whether a given set of polynomials is in the form of a Gröbner basis or not.

Lemma 1. *Given a finite set* $G \in \mathbb{Z}_2(x_1, \ldots, x_n)$*, suppose that we have* $p, g \in G$ *such that* $\mathrm{LCM}(\mathrm{lm}(p), \mathrm{lm}(g)) = \mathrm{lm}(p) \cdot \mathrm{lm}(g)$*. In other words, the leading monomials of* p *and* g *are relatively prime. Then* $\mathrm{Spoly}(p, g) \xrightarrow{G}_+ 0$ *[26].*

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

Definition 15 (Gröbner Bases)**.** *A finite subset* $G = \{g_1, \ldots, g_{\hat{s}}\}$ *wrt. a monomial order* $\prec$ *of an ideal* I *is said to be Gröbner basis of* I *if* $V(G) = V(I)$ *and all leading monomials in* G *are relatively prime.*

In [68], the monomial order $\prec$ following the reverse topological order of the circuit has been shown to be effective in modeling the circuit as Gröbner basis in linear time. Because of this monomial order, the time complexity of the Gröbner bases algorithm has been bypassed, and utilizing the IMT procedure becomes computationally feasible.

A given ideal may have different Gröbner bases, where one set of bases can be reduced wrt. a monomial ordering to other bases. The thesis defines G as *strong Gröbner bases* of ideal I, if for any term of $p \in I$ there exists a polynomial $g \in G$ satisfying that $lt(g)$ divides this term of p. Note that for fields, any non-zero coefficient of a term is invertible, it is easy to verify that in this case, every Gröbner basis is a strong Gröbner basis, however, this does not hold in general for rings, hence the utilized modeling method restricts the values of coefficients of all $lt(g_i) \in G$ to the set $\{-1, 1\}$, as introduced in Subsection 2.1.5. Strong Gröbner bases enable the polynomial division operation required by the IMT.

Definition 16 (Polynomial Division)**.** *A* polynomial division *of two polynomials* p *and* g *denoted as* $p \xrightarrow{g}_+ r$ *is performed as* $r = p - \frac{c \cdot M}{\mathrm{lt}(g)} \cdot g$*. If a non-zero term* $c \cdot M$ *of* p *is divisible by the leading term of* g*, then* p *reduces to* r *modulo* g*. Similarly,* p *can be reduced (divided) wrt. a set of polynomials* G *to obtain a remainder* r*, denoted* $p \xrightarrow{G}_+ r$*, such that no term in* r *is divisible by the leading term of any polynomial in* G*.*

The polynomial division in this form is not applicable, if G is not a strong Gröbner basis. Because at this case the $\mathrm{lc}(g)$ may not equal to one, note that $\mathrm{lt}(g) = lc(g) \cdot lm(g)$, therefore, the coefficients of the resulting polynomial r will not be in the integer ring.

The polynomial division $p \xrightarrow{g}_{+} r$ can be seen as the *substitution* of a variable x in p with the tail terms of g using a *rewrite rule*, whereas x is also the leading monomial of g ($lm(g) = x$).

Definition 17 (Rewrite Rule)**.** *Let $g = -lm(g) + tail(g)$. The rewrite rule corresponding to g substitutes the $lm(g) = x$ in a polynomial $p \in \mathbb{Z}_2(x_1, \ldots, x_n)$ by the $tail(g)$, which is denoted as $lm(g) \rightarrow tail(g)$. If g is a monomial, then the right-hand side of its rule is 0. The thesis refers to applying the rewrite rule as the substitution of variable x.*

Example 6. *Let $p := x_4x_3 + x_1$ and a polynomial $g := -x_4 + x_2x_1$, then $r = p - \frac{x_4x_3}{-x_4}g = x_3x_2x_1 + x_1$, where the polynomial division substitutes x_4 in p with x_2x_1.*

In the IMT procedure, as shown in Algorithm 1, given a specification (or relationship) polynomial p_r and a circuit model in form of Gröbner bases G, p_r is divided in every iteration by some polynomial $g \in G$. The division (substitution) iterations are executed according to a certain order, the *substitution order*. This order is crucial to cancel the nonlinear terms before the blow-up of their intermediate sizes. In [23, 39], the substitution order follows the reverse topological order of the circuit variables, in addition to the fanouts of the gates—variables that have the same level and depend on common inputs (fanouts) must follow each other in the substitution.

Example 7. *Following Example 5, the extracted algebraic model is a Gröbner basis, hence the ideal membership testing of p_r can be applied. The substitution order follows the reverse topological order of the circuit:*

$$
\begin{aligned}
p_\mathrm{r} &\xrightarrow{g_1}_{+} -s + 2x_4x_3 - 2x_4 - 2x_3 + x_3 + x_2 + x_1 \\
&\xrightarrow{g_2}_{+} 2x_4x_3 - 2x_4 - 2x_3 + 2x_1x_3 - x_1 + x_2 + x_1 \\
&\xrightarrow{g_3}_{+} 2x_3x_2x_3 - 2x_3 - 2x_2x_3 + 2x_1x_3 - x_1 + x_2 + x_1 \\
&\xrightarrow{g_4}_{+} 2x_2x_3x_2x_1 - 2x_2x_3 + 2x_1x_3 - x_1 - 2x_2x_1 + x_2 + x_1 \\
&\xrightarrow{g_5}_{+} 2x_1x_3 - x_1 + 4x_3x_2x_1 - 2x_3x_1 - 2x_3x_2 - 2x_1x_2 + x_2 + x_1 \xrightarrow{g_6}_{+} 0.
\end{aligned}
$$

Since the final division result is 0, it is proven that the circuit under verification satisfies the function specification p_r.

Algorithm 1 IMT Procedure

Require: Equivalence relationship polynomial p_{r}, circuit polynomials $G = \{g_1, g_2, \ldots, g_s\}$

Ensure: Remainder r is equal to zero

1: $V \leftarrow OrderedPolynomialVariables(p_{\mathrm{r}}, G)$ { Substitution ordering}
2: $r \leftarrow p_{\mathrm{r}}$
3: **for** i **in** 0 **to** $|V| - 1$ **do**
4: **if** $V[i] \notin PrimaryInputs$ **then**
5: Choose $g_t \in G$ such that $\mathrm{lm}(g_t) = V[i]$
6: $r \xrightarrow{g_t}_{+} r$
7: **end if**
8: **end for**

The thesis leverages also another property of the Gröbner bases theory to simplify and preprocess the ideal of the circuit model. For a given ideal (set of polynomials), the theory offers a canonical representation for the ideal called *reduced Gröbner bases*.

Definition 18 (Reduced Gröbner Bases)**.** *A reduced Gröbner basis is Gröbner basis G for a polynomial ideal I, such that for all $g_i \in G$, no term in g_i is divisible by the leading term $lt(g_j)$ for all $i \neq j$.*

Lemma 2. *Let $I \neq 0$ be a polynomial ideal. Then, for a given monomial ordering $\prec$, I has a unique reduced Gröbner basis [26].*

In Chapter 5, the thesis utilizes the uniqueness property of the reduced Gröbner bases for computing canonical polynomials and for checking the *equality of ideals*. As a consequence of Lemma 2, once reduced Gröbner bases can be effectively computed for two given ideals, it is effortless to deduce that the ideals are equal if and only if they have the same reduced Gröbner basis. The reduced Gröbner basis can be computed by eliminating variables of the given Gröbner basis according to a specific substitution order. By eliminating variables recursively new Gröbner bases are generated until an ideal that satisfies conditions of the reduced Gröbner basis is derived.

Example 8. *Consider two Gröbner bases G_1 and G_2 modeling the same Boolean function $x_o = x_1 \oplus x_2 \oplus x_3$, where $G_1 = \{-\hat{v}_1 - 2x_2x_1 + x_2 + x_1,\ -x_o - 2\hat{v}_1x_3 + \hat{v}_1 + x_3\}$ and $G_2 = \{-v_1 - x_2x_1 + x_1,\ -v_2 - x_2x_1 + x_2,\ -x_3 - v_2v_1 + v_2 + v_1,\ -v_4 - v_3x_3 + v_3,\ -v_5 - v_3x_3 + x_3,\ -x_o - v_5v_4 + v_5 + v_4\}$. Eliminating (substituting) the variable $\hat{v}_1$ by the rewrite rule from G_1 leads to the reduced Gröbner basis $rG_1 = \{-x_o + 4x_1x_2x_3 - 2x_3x_2 - 2x_3x_1 - 2x_2x_1 + x_3 + x_2 + x_1\}$, while by*

eliminating the variables of G_2 according to the order $v_5 > v_4 > v_3 > v_2 > v_1$, the reduced Gröbner basis $rG_2 = \{-x_o + 4x_1x_2x_3 - 2x_3x_2 - 2x_3x_1 - 2x_2x_1 + x_3 + x_2 + x_1\}$ is derived. Because $rG_1 = rG_2$, the equality of the two ideals is proved.

2.3 FORMAL VERIFICATION OF ARITHMETIC CIRCUITS

Formal verification of floating-point and integer arithmetic has been the subject of extensive investigation in academia and industry, both to verify the correctness of the computation and control aspects. Verification of arithmetic circuits is expensive due to large and complex hardware given the inherent difficult nature of the computation. Because of this, most if not all of recent proposed works decompose the problem manually or through specific case splitting approaches and apply different techniques to verify the decomposed units. These methodologies require high-level expertise and a perfect understanding of the design. There is a large body of work from Intel outlining methodologies that combine automated model checking and theorem proving to verify FPUs. As a paradigmatic example of using this combination, a divider circuit is verified by constructing an inductive proof using a theorem prover to verify that certain invariants are maintained, while the circuit-level details are checked by a model checker [55, 76]. The most recent exposition of these works [61] require implementation-specific tedious manual effort to make the verification process complete. In a similar fashion, AMD leverages the ACL2 theorem prover and a model checker to verify formally FPUs with manually-guided proofs [88]. A higher automated methodology from IBM [54] combines case splitting, multiplier isolation, and automatic equivalence checking, to make the verification of fused-multiply-add FPUs tractable.

In the following, the thesis describes briefly architectures of multipliers that have been verified by proposed techniques of the thesis, in addition to the main specifications of FPUs as defined in the latest IEEE Standard for floating-point arithmetic [50]. Also, it classifies formal verification methodologies for arithmetic circuits into two directions: 1) methodologies utilize equivalence checking and 2) methodologies combine theorem proving with automated checkers.

2.3.1 *Multiplier Architectures*

The Multiplication function involves two basic operations to generate and accumulate partial products. There are three types of high speed multipliers [62]

which are *sequential multiplier*, *parallel multiplier* and *array multiplier*. However, the thesis focuses on parallel and array multipliers since they are the most common types. The parallel multiplier generates partial products in parallel and accumulates them using a fast multi-operand adder, while the array multiplier consists of identical cells which are typically full adders or carry-save-adders that are connected in a way that reduces the delay of the longest path in the circuit (*logic depth*).

To design a high speed multiplier, the designer reduces the number of partial products and/or accelerates their accumulation. This is done by optimizing at least one of the three main parts of the multiplier circuit:

1. The first part is the generation of partial products. They can be produced simply from the logical AND of the multiplicand with the multiplier. However, *Booth's algorithm*—in particular Radix-4 modified Booth recoding—is utilized to reduce the number of partial products.

2. The second is the multi-operands adder tree which sums up partial products to two arrays of output bits. There are many types of trees that make a trade-off between the overall delay and the wiring complexity (number of needed wires or tracks) of the circuit. For instance, *Wallace tree* is known for its optimal computation time, but it suffers from a large number of wires, in contrast, *balanced delay tree* requires a smaller number of wiring tracks but has a higher overall delay compared with the Wallace tree.

3. The third part is the last stage two-operands adder which is given outputs of the adder tree and generates a multi-bits array as an output of the multiplier circuit. The straightforward implementation of the final adder is the *ripple carry adder* which can be thought of as an array of full adders, where the carry-out of the ith full adder is fed to the carry-in of the (i+1)th full adder. Another group of adders which has less delay than the ripple carry adder is called *Parallel Prefix Adders* (PPAs) since the adders reduce the amount of time required to determine carry bits. Types of PPAs diverge in the logic depth, the gates fan-out, and the area.

The thesis categorizes the integer multiplier architectures according to 1) the type of the partial products generator, 2) the partial products accumulator, and 3) the last stage adder. In the experiments, the verified integer multipliers circuits are combinations of the following:

1. Two types of partial products generators, namely *Simple Partial Products* (SP), and *Booth Partial Products* (BP).

2. Multiple types of partial products accumulators which are *Array* (AR), *Wallace Tree* (WT), (4,2) *Compressor Tree* (CT), *Redundant Binary Addition Tree* (RT), and *Dadda Tree* (DT).

3. The chosen types of the last stage adder are *Ripple Carry Adder* (RC) as well as PPAs which are *Carry Look-Ahead Adder* (CL),*Brent-Kung Adder* (BK), *Kogge-Stone Adder* (KS), and *Hans-Carlson Adder* (HC).

These multiplier benchmarks are named according to their architecture features. For example, a circuit with simple partial products, a Wallace tree as partial products accumulator, and a ripple carry adder as last stage adder will be labeled by SP-WT-RC. The integer multiplier benchmarks as HDL are generated using the online tool *Arithmetic Module Generator* [2].

2.3.2 *Floating-Point Specification*

Floating-point (FP) numbers overcome the limitation of fixed-point integers, they can provide a vast range of numbers. Fixed-point representations have a fixed window of expressible numbers, they cannot represent very large and very small numbers at once. In contrast, FP maps the infinite range of real numbers by a finite subset with limited precision. Despite its range, FP has a serious drawback: the represented numbers are inaccurate—they are approximated. Because there are many possibilities to approximate a real number, IEEE standard for FP is proposed. The arithmetic standard [50] is the most common representation for real numbers in the state-of-the-art microprocessors. It provides a standard method for computation with FP numbers that will yield the same result regardless of the implementation—the results of the computation including errors and error conditions will be identical given the same input data.

Definition 19 (Binary Floating-Point)**.** *A binary floating-point number is defined by a triple* (s, ex, m) *with sign bit* $s \in \{0, 1\}$*, exponent* $ex \in \mathbb{Z}$*, and significand (mantissa)* $m \in \mathbb{R}_{\geq 0}$*. The value of the number is calculated as* $(-1)^s \cdot 2^{ex} \cdot m$*.*

The standard determines a set of finite binary FP numbers representable within a particular format by the integer parameters: $p =$ the number of bits in the significand m (*precision*), $emax =$ the maximum exponent ex, and $emin = 1 - emax =$ the minimum exponent ex. Within each format, ex is any integer in the range $emin \leq ex \leq emax$ and m is a number represented by a bit string of the form $b_0 \cdot b_1 b_2 \cdots b_{p-1}$ where $b_i \in \mathbb{B}$, therefore, $0 \leq m < 2$.

Table 1: Binary Format Parameters

Parameter	binary16	binary32	binary64	binary{k} at $k \geq 128$
k	16	32	64	multiple of 32
p	11	24	53	$p = k - round(4 \cdot log_2(k)) + 13$
$emax$	15	127	1023	$2^{(k-p-1)} - 1$

Representations of FP numbers in the binary interchange formats are encoded in k bits allocated as follows: 1-bit for sign, w-bit for biased exponent $E = ex + bias$, and $(t = p - 1)$-bit for trailing significand field $b_1 b_2 \cdots b_{p-1}$; the leading bit of the significand b_0 is implicitly encoded in the biased exponent E. The values of w, t, and $bias$ for binary formats are calculated given the value of k and p based on the following equations: $w = k - p$, $t = p - 1$, and $emax = bias = 2^{(w-1)} - 1$. The standard defines binary formats of widths 16, 32, 64, and 128 bits, and in general for any multiple of 32 bits of at least 128 bits. The parameters p and k for every format width are shown in Table 1.

According to the standard, every operation on FP numbers is performed as if it first produced an intermediate result that is correct to infinite precision, and then rounded this intermediate result based on given rounding modes to fit in a destination binary format. In other words, if the result of an arithmetic operation does not fit in the precision p or the range of the exponent e, the result is rounded according to a given round mode. The standard supports four rounding modes for binary FP:

1. roundTiesToEven delivers the FP number nearest to the infinitely precise result, if the two nearest FP numbers are equally near, the one with an even least significant bit is delivered;

2. roundTowardPositive delivers the FP number closest to and no less than the infinitely exact result;

3. roundTowardNegative delivers the FP number closest to and no greater than the infinitely exact result;

4. roundTowardZero delivers the FP number closest to and no greater in magnitude than the infinitely exact result.

The thesis evaluates the performance of its proposed techniques by verifying integer multipliers and FP multipliers. The FP multiplication opera-

tion computes $C = A \cdot B$ for two FP operands $A = (-1)^{s_a} \times 2^{e_a} \cdot m_a$ and $B = (-1)^{s_b} \cdot 2^{e_b} \cdot m_b$. s_a denotes the sign, e_a the exponent, and m_a the significand including the implicit bit of the operand A (similarly for B and C). The operation can be defined as $s_c = s_a \oplus s_b$ and $2^{e_c} \cdot m_c = \text{RND}\ (2^{e_a+e_b} \cdot m_a \cdot m_b)$. RND is the round function that fits the exact result of $(2^{e_a+e_b} \cdot m_a \cdot m_b)$ in the FP number $(2^{e_c} \cdot m_c)$.

Such arithmetic specification could be formulated mathematically for theorem proving and model checking [53, 70] or as a HDL-based reference model for equivalence checking as in [54] and also in this thesis.

2.3.3 *Equivalence Checking*

Equivalence checking verifies that given two designs with the same corresponding inputs, the corresponding outputs of these designs are always equal under all input assignments. Equivalence checking is performed by building a so-called miter, where the two compared designs are modeled using the same description method and combined in one design. A miter is constructed by adding 2-input XOR gates on top of corresponding outputs of compared designs together with connecting the outputs of these XOR gates to a large OR gate. The miter has a single output which is the output of the OR gate, the output is equal to one if there is an assignment causing a mismatch between at least one output pair.

Modern equivalence checking tools are capable of verifying designs with millions of gates in very short times. The great success of equivalence checking is based on exploiting structural similarities between the two compared circuits. Structurally similar circuits contain a lot of internal nodes implementing equivalent circuit functions. These internal equivalences sometimes called cut points [65] are deduced automatically and leveraged efficiently to decompose the verification problem into smaller ones [11, 65, 72]. This process is performed typically over the AIG representation of the miter.

Checking the equivalence between two combinational designs or two sequential designs with the same state encodings is named *Combinational Equivalence Checking* (CEC). Another type of equivalence checking is called *Sequential Equivalence Checking* (SEC) [5, 38], it compares two sequential designs wherein there are no one-to-one correlations between some or all of their states. Generally, SEC requires analysis of the sequential behavior of compared designs so that it comes with greater computational expense than CEC.

Both types of equivalence checking cannot deal with designs that have few internal equivalences. This problem occurs especially for arithmetic circuits since one arithmetic function can be implemented in many different ways [96], in particular integer multipliers implemented at the gate level. Because of this, there is no general automated solution that can be applied on all types of circuits. The thesis classifies equivalence checking approaches into two groups based on the types of the checked circuits: 1) integer multipliers, and 2) FPUs.

2.3.3.1 *Integer Multipliers*

Without any information on the high-level structure of the netlists most equivalence checkers fail to verify multiplication circuits. For this problem, several approaches have been proposed in the past. However, most of these approaches are not scalable, cannot be applied on all types of multiplier architectures, or they fail if the circuit does not represent a correct multiplier function. The existing approaches can be divided mainly into three categories: 1) decision diagrams, 2) structural methods, and 3) approaches based on special arithmetic properties of the realized function.

Decision diagrams provide a canonical representation for netlists of the design under verification (DUV) and its reference implementation (RI). The equivalence checker exploits this property to check that the resulting decision diagrams are identical. Unfortunately, many popular data structures like BDDs [15] have an exponential size for multiplication. *BMDs [17] can represent the word level multiplier function in a compact way. However, the exponential blowup can still occur for incorrect designs or during the construction of the *BMD from a bit-level circuit [47].

The second direction in the classification is based on structural methods. An approach in this direction has been proposed in [96]. It extracts adder structures from bit-level netlists and builds full adder networks. This approach has mainly two drawbacks: First, it makes assumptions about the internal structure of the circuits that are not fulfilled by all multiplier architectures [97]. Second, the approach fails to build the full adder networks if the circuit does not represent a correct multiplier, leading to an inconclusive result. Nevertheless, this technique achieves promising results—it allows to verify a 48-bit multiplier in about 30 minutes.

The third direction exploits arithmetic properties of the multiplier function to build a miter with many internal equivalences. The first approach in this context has been proposed by Fujita [42]. It is based on the fact that any function satisfying the recurrence relation $(X + 1) \cdot Y = X \cdot Y + Y$ is a multiplication.

However, the original approach by Fujita does not scale, and it cannot verify multipliers larger than 16 bits. A related approach [21] is based on case splitting by forcing a bit of one of multiplier operands to be zero and summing partial products that belong to this bit outside the multiplier. The problem with this idea is that the similarity of the nets inside the miter structure depends on the order of the partial products of the DUV, therefore, this approach only works for specific implementations of multipliers.

2.3.3.2 *FPUs*

In [54, 102], two similar combinational equivalence checking methodologies have been proposed to verify FP of a *Fused-Multiply-Add* (FMA), they are also applicable to FP of adders and multipliers. FMA computes the function $A \cdot B + C$ on FP operands A, B, and C, while multiplication $A \cdot B$ and addition $A + B$ operations are computed using FMA as $A \cdot B + 0$ and $A \cdot 1 + B$, respectively. In addition to the multiplier block, shifters of the FMA circuit are major challenges for the state-of-the-art equivalence checkers. FMA contains two shifters: the alignment shifter that aligns the addend C to the result of product $(A \cdot B)$ and the normalization shifter that eliminates leading zeros in the intermediate result before rounding. Each of these building blocks leads to exponential runtimes with SAT and exponential sizes with BDDs.

In [54], the RTL implementation of the FPU is compared against a simple behavioral model described in an HDL language, while in [102], it is compared against a C/C++ model. In both methodologies, the overall equivalence checking problem is split into subproblems to circumvent difficulties posed by shifters. In this way, the shift amounts of shifters are restricted to small ranges, causing them to collapse into simple wires. To circumvent the difficulties posed by the multiplier, it is removed from the cone-of-influence of the compared FPUs and verified independently by one of the approaches described in the previous subsection.

In [63], SEC is leveraged to verify the control aspects of FPU, whereas CEC can verify only data-path aspects, as in [54, 102]. This is achieved by: 1) assuming that the FPU produces the right result without accounting for control aspects such as interactions between instructions and resource conflicts; and 2) comparing the FPU against itself under different conditions. The RI of the miter is an instance of the FPU that is given a single random instruction in an empty pipeline, while the DUV will be another instance of the FPU that is fed the same instruction as a part of a sequence of random instructions. The miter checks only the results of the given instruction between RI and DUV.

This setup enables control features in the DUV to be invoked, and if the given instruction interferes with the execution of other instructions, the miter will announce a mismatch. This special set-up allows leveraging structural similarities between the RI and the DUV (they are just two instances of the same design) to make the SEC applicable which otherwise would not be.

2.3.4 *Theorem Proving*

Typically, formal verification of FP circuits has been performed by combining theorem proving with automated checking techniques such as model checking and equivalence checking. In the following, a brief description about theorem proving is given, then the integration between theses formal verification techniques to verify FP circuits is presented.

Theorem proving [49, 52] constructs an inductive formal proof that is guided manually for the correctness of the DUV. This proof is assisted by theorem provers which provide libraries of large numbers of basic theories and lemmas, e.g., a bit vector library. Also, they can integrate features such as an expressive specification language, a functional programming language, and powerful deductive techniques. These features allow expressing the specification as a set of lemmas, modeling of the DUV as a composition of recursive functions, and asserting the correctness of the DUV with respect to the specification lemmas by deductive reasoning. A proof for specification lemmas (properties) is obtained by combining a set of previously proved lemmas that together imply the desired property. Each lemma holds relative to some subset of previously proved lemmas; this prior knowledge is utilized to prove the new lemma. A given lemma usually focuses on one aspect of the design. Typically, early lemmas describe variable domains, properties about local data structures of the design, while later lemmas address more global aspects of the design. Utilizing the prior information helps the later lemmas to be proved fairly easy since it causes a reduction of the proof complexity. When it is too difficult to prove a lemma directly, a skilled human verifier typically searches for additional supporting lemmas that make it feasible to prove such a new one.

The strength of the combination between theorem proving, model checking and equivalence checking, is in the use of automated checkers to hide the bit-level details of the circuit and resolve the control as well as scheduling constraints of the design, while theorem proving overcomes the scalability limitation of the automated checking by enabling the construction of a high-level mathematical proof, whereas most of FPU specifications including the IEEE

Standard are based on real numbers, not Booleans. This combination bypasses the scalability drawback of bit-level proof solvers such as BDD and SAT which are used within automated checking techniques. However, this combination is still requiring manual interaction and high understanding of the FPU design to construct a tedious mathematical proof that can be checked by a theorem prover. Because of this human intervention, such a verification process suffers also from unintentional human faults in the constructed proof. This concept has been utilized for more than two decades [52, 56, 74, 76, 79, 82, 88, 90, 91] for verifying FPUs and integer multipliers. An implementation of the concept is the formal verification framework being used at Centaur [91], it ties together the ACL2 theorem prover [59] with non-commercial automated tools: ABC equivalence checker [13], MINISAT solver [37] and IC3 model checker (PDR) [36], whereas the verification engine is built upon the AIG and BDD symbolic models.

Another example to illustrate this concept is the approach proposed by IBM [79] to verify the RTL of Booth integer multipliers. It takes two operands X and Y, producing later after n-cycles two bit-vectors Sum and Carry such that Sum + Carry = X · Y. The correctness of this design is encoded by the following ACL2 theorem:

```
(defthm multiplier-correct
(implies
(and (integerp n)
(<= 7 n))
(equal (bv+ Sum Carry)
(bv (* (bv-val X) (bv-val Y)) L)
)))
```

The theorem states that the product of X times Y is equal to the addition of Sum and Carry. Because the correct output of the multiplier starts to stream out after 7 cycles of initialization and filling the pipeline, the condition (<= 7 n) is added. The operation bv+ retrieves a bit vector representing the sum of the two arguments. The function bv-val returns the integer value of a bit-vector, while (bv v L) returns a bit vector of the length L representing a value v.

The overall proof strategy is based on decomposing this final theorem into simple properties that can be checked automatically by a model checker, while the theorem prover is utilized to prove the consistency between the decomposed properties and the final theorem. First, the final theorem is reduced into two major lemmas: 1) the correctness of the Booth encoder, and 2) the correctness of

the subsequent compressions stages. The Booth encoder lemma states that the addition of all the bit-vectors coming out of the Booth encoder is equal to the product of the input operands. The second lemma states that the summation is preserved during the compression tree. Combining these two major lemmas leads to prove the correctness of the multiplier.

As these major lemmas are described over integer operations while the multiplier implementation is described at bit-level, three ACL2 models of the Booth encoder are created named high-level model, low-level model, and the bit model. The main difference between the low-level model and the bit model is that the low-level model uses arithmetic and propositional logic operations, while the bit model is purely constructed of propositional logic operations. To verify that the DUV satisfies the major lemma of Booth, first, the theorem prover verifies the high-level model against this major lemma; then, it proves the equivalence between high-level and low-level models; finally, it compares the low-level model against the bit model. By performing these steps, the major lemma of Booth is decomposed into simple lemmas (properties) which build the bit model wherein each property states the correctness of generating one Booth vector. Thereby it is feasible to check independently that the DUV satisfies each property in the bit model using the model checker.

To verify the implementation of the compression tree which sums the generated Booth vectors into two bit-vectors, the equivalence of its major lemma is checked by the theorem prover against a set of properties that describe functions of each carry-save-adder in the compression tree, while each individual property is checked by the model checker wrt. the DUV in a practical time.

In IBM, the entire verification efforts to perform this proof for a specific multiplier design required about a month, 21 eight hour work days of human effort from a single experienced ACL2 user. About one-third of this time was spent finding properties that could be verified by the model checker together with writing the necessary configuration files. The remaining two-third of the time was spent developing the necessary ACL2 proof, while the proof itself requires about 50 minutes to run.

3

RECURRENCE RELATIONS: SCALABLE VERIFICATION OF MULTIPLIERS

Although a lot of effort has been spent on verifying arithmetic designs, it is still a problem that has no general robust automated solution. One major challenge is verifying large scale multiplier circuits. For this purpose, this chapter revisits the idea of using functional properties of the multiplication function, which can be expressed by recurrence equations. Then, instead of proving the equivalence of the implementation and a specification, the verification task is to show that the implementation satisfies the recurrence equation. We propose an approach which makes this circuits. Based on a combined add/multiply recurrence equation, we can make efficient use of case splitting wrt. the partial products of the multiplier. As a result, the verification problem is split into simpler cases such that only a small part of the multiplier will be checked in every case, thereby avoiding redundant checks among the cases. In every decomposed case, the generation and the addition of one partial product of the multiplier are checked. Since the multiplier is an addition tree that adds all partial products, checking the correctness of generating and adding every partial product in the multiplier by the proposed case splitting leads to the verification of the complete multiplier. As the multiplier size increases, the number of cases will increase, while the complexity to check one case will remain almost the same. The proposed approach is able to verify a multiplier at the gate-level without any information about its high-level specification or the internal structure of the netlist. In addition, it is a general technique that can be applied to various different architectures. Overall, it allows verifying large scale multipliers in practical time. The experiments show the ability of the proposed approach to verify 128-bit multiplier.

In the summary, the main contributions of this chapter are:

1. Explaining the theoretical background of equivalence checking based on recurrence relations, which has not been discussed in detail before.

2. Enhancing the scalability of an equivalence checking technique that does not require a golden reference, whereas such references are not available in many practical cases.

3. Proposing a new automated decomposition method for the verification problem of the multiplier which overcomes the exponential complexity of this problem.

3.1 EQUIVALENCE CHECKING BASED ON RECURRENCE RELATIONS

In classical equivalence checking, the *Design Under Verification* (DUV) and the Reference Implementation (RI) are combined in one circuit called miter by XOR-ing every output pin of the DUV with the respective output pin of the RI and computing the OR of these functions. An efficient miter should have such many similar nets, so that the equivalence checking problem can be partitioned into less complex sub-problems.

In this section, we review an equivalence checking that builds the RI from the DUV itself, which is referred as *Equivalence Checking based on Recurrence Relations* (ECRC). The technique does not need a golden reference model and does not require any knowledge about the internal architecture of the DUV. In the same time, it offers a miter with a proper number of equivalences. The ECRC technique is utilized to verify functions that can be defined by recurrence relations. These functions are also known as *primitive recursive* (p.r.) functions [27]. In the following, we briefly discuss p.r. functions, before showing how to exploit the recurrence relations for equivalence checking. Finally, we give an overview about Fujita's approach [42] which leverages an equivalence checking based on a recurrence relation.

The p.r. functions are defined by the following theorem, as stated in [27]:

Theorem 2. *Given p.r. functions* $g \in \mathbb{N}^k \to \mathbb{N}$ *and* $h \in \mathbb{N}^{k+2} \to \mathbb{N}$*, there is a* ***unique*** $f \in \mathbb{N}^{k+1} \to \mathbb{N}$ *satisfying:*

$$f(0, \vec{Y}) = g(\vec{Y})$$

$$f(X+1, \vec{Y}) = h(f(X, \vec{Y}), X, \vec{Y})$$

for all $\vec{Y} \in \mathbb{N}^k$ *and* $X \in \mathbb{N}$*.*

Based on this theorem, a unique p.r. function f is defined, if it can be constructed from other p.r. functions g and h using a composition and a primitive recursion. Let $\vec{Y} = (Y_0, Y_1, \cdots, Y_{k-1})$ and $X, Y \in \mathbb{N}$. The basic p.r. functions are: 1) the constant zero $C(\vec{Y}) = 0$, 2) the projection function $P_i(\vec{Y}) = Y_i$, and 3) the successor function $S(X) = X + 1$. Known examples for p.r. functions are addition and multiplication. The addition function $Add(X, Y)$ is defined

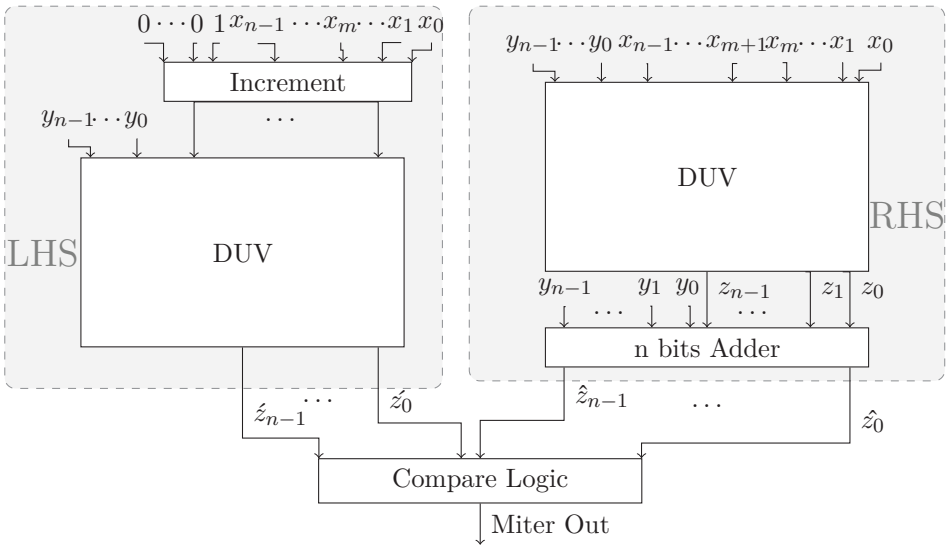

Figure 12: Miter of Multiplier using Fujita's Approach

according to Theorem 2 by the relations $Add(0, Y) = Y$ and $Add(X + 1, Y) = S(Add(X, Y))$. Note that the first relation is given in terms of a projection function, while the second uses the successor function, whereas both are basic p.r. functions. This implies that Add is a p.r. function as well. Based on this, the multiplication function $Mult(X, Y) = X \cdot Y$ is uniquely defined by $Mult(0, Y) = 0$ together with $Mult(X + 1, Y) = Add(Mult(X, Y), Y)$, thereby it is also a p.r. function.

The uniqueness of a p.r. function thus defined gives rise to the ECRC technique. Given some implementation F of a p.r. function f, we can check that F correctly realizes f by checking that it satisfies the recurrence relations of f. In case of the multiplication function, if we succeed to show that F satisfies the relations $F(0, Y) = 0$ and $F(X + 1, Y) = Add(F(X, Y), Y)$ for all inputs X and Y, this implies that F is indeed a multiplication. For most p.r. functions, checking the first relation is relatively easy, while the second relation is verified by building a miter according to the recurrence relation and using standard equivalence checking techniques to show that both sides of the equation are equal for arbitrary inputs. The checking of both relations is referred as the ECRC technique.

An example of ECRC has been proposed by Fujita in [42]. The approach verifies a multiplier circuit by checking the equivalence of both sides of the recurrence relation:

$$Mult(X+1,Y) = Mult(X,Y) + Y.$$

The resulting miter structure is shown in Figure 12. An implementation of the *Left Hand Side* (LHS) of the equation is built by adding an increment circuit to the first operand X of the DUV, while the *Right Hand Side* (RHS) is implemented by adding the second operand Y to the output Z. The equivalence checking compares the two circuits in order to prove that the DUV is indeed a multiplier.

Although the two sides use the same circuit for multiplication, they have different inputs: $(X+1,Y)$ on the LHS and (X,Y) on the RHS. Therefore, the equivalence checker is not able to find enough internal equivalence points. Fujita overcomes this problem by splitting the checking process into a series of sub-problems. The case splitting is designed based on the least bit x_m of X that has a zero value in the increment function $(X+1) = ((2^{n-1}x_{n-1} + 2^{n-2}x_{n-2} + \cdots + 2x_1 + x_0) + 1)$. Let x_m is equal to zero in the bit vector X, incrementing X by one inverts the value of x_m from zero to one iff all lower bits $x_i, 0 \leq i \leq m-1$ are equal to 1, besides the inversion of x_m, all these lower bits will be inverted from one to zero, e.g., $((2^{n-1}x_{n-1} + \cdots + 2^{m+1}x_{m+1} + 2^m \cdot 0 + 2^{m-1} + \cdots + 4 + 2 + 1) + 1) = (2^{n-1}x_{n-1} + \cdots + 2^{m+1}x_{m+1} + 2^m)$. Fujita's approach leverages this feature of the increment function to split the equivalence checking problem into simpler cases, in each case m the increment circuit is replaced by simple inverters for the $m+1$ lower bits of X together with assigning one to all inputs x_i for $i < m$ and zero for the variable x_m; the higher bits x_j, for $j > m$ will not be modified. This simplification offers internal equivalence points since every case m creates a similarity between upper parts of the LHS and the RHS of the miter, where most of the input bits (i.e., x_i for $i > m$) to these upper parts are the same in both sides. However, for larger multipliers beyond 16 bit, this simplification of the increment circuit does not lead to sufficient similarities that enable a scalable verification.

In the next section, we introduce a new approach which increases the scalability of the ECRC technique to verify multipliers larger than 16 bits.

3.2 CHECKING PARTIAL PRODUCT APPROACH

The ECRC has a very useful feature that is missed in classical equivalence checking techniques, it does not assume that there is a golden reference that is compared against the DUV. Also it does not require a structural knowledge about the DUV that helps to find the proper reference that have many structural similarities with the DUV. For these reasons, we are interested in enhancing the scalability of the ECRC. We propose an approach that exploits features of the ECRC for the verification of large scale integer multipliers.

In following, after some basic definitions of the multiplication function and the combined add/multiply function, we give an overview of this proposed approach. Then, we present the theoretical part of the approach. Finally, its implementation will be introduced.

3.2.1 *Basic Notions*

We denote an integer multiplier as $Z = X \cdot Y$, X and Y are the integer operands of the multiplier, Z is the integer result of the multiplier. These integer variables will be represented as vectors of Boolean variables such that $X = \sum_{i=0}^{n-1} 2^i x_i$, $Y = \sum_{i=0}^{n-1} 2^i y_i$, and $Z = \sum_{i=0}^{2n-1} 2^i z_i$, where n is the size of each operand of the multiplication function and 2^i is the weight of each Boolean variable. The multiplier $Z = X \cdot Y$ can be expressed on the bit-level as:

$$\sum_{i=0}^{2n-1} 2^i z_i = \sum_{i=0}^{n-1} 2^i x_i \cdot \sum_{i=0}^{n-1} 2^i y_i.$$

Typically, the multiplication of two operands is performed by generating partial products which are then summed using an addition tree as shown in Figure 13.

The *partial product* is a bitwise multiplication $pp_{ci,ri} = y_{ci-ri} \cdot\ x_{ri}$, where ri and ci are the row/column indices of the partial product in the addition tree. The $pp_{ci,ri}$ has a weight 2^{ci} which is the product between weights of x_{ri} and y_{ci-ri}, e.g., Figure 13 shows the partial product $pp_{4,4} = y_0 \cdot x_4$ which has the weight 2^4. For an n-bit multiplier, we define that

$$pp_{ci,ri} = \begin{cases} y_{ci-ri} \cdot x_{ri} & 0 \leq ci - ri \leq n-1 \\ 0 & Otherwise. \end{cases}$$

A *partial products generator* is a part of the multiplier which produces partial products by multiplying each bit of the first operand by all bits of the second operand in that the summation of generated partial products satisfies the equation

$$\sum_{ci=0}^{2n-2}(2^{ci}\cdot\sum_{ri=0}^{n-1}pp_{ci,ri})=\sum_{i=0}^{n-1}2^{i}x_{i}\cdot\sum_{i=0}^{n-1}2^{i}y_{i}.$$

The *addition tree* is the multiplier part which compresses partial products to generate the multiplier result Z. The function of the addition tree is defined by the equation

$$\sum_{i=0}^{2n-1}2^{i}z_{i}=\sum_{ci=0}^{2n-2}(2^{ci}\cdot\sum_{ri=0}^{n-1}pp_{ci,ri}).$$

Note that $pp_{ci,ri}$ is equal to zero, if $ci-ri<0$ or $ci-ri>n-1$. The tree has two indices ri and ci, where ri is the index of the tree rows, while ci is the index of the tree columns. As shown in Figure 13, partial products are ordered in the tree such that those of the same weight 2^{ci} belong to the same column ci. The output bit z_{ci} belongs also to the column ci since it is of weight 2^{ci}. This description of the addition tree allows to define it as sets, where each set $CE_{ci}=\{pp_{ci,0},pp_{ci,1},\cdots,pp_{ci,n-1}\}$ consists of partial products of a column ci in the tree, together with relating each set CE_{ci} with an output variable z_{ci}.

The *addition tree equation* defines the addition of the elements in CE_{ci} as well as the function of its related output variable z_{ci}. The addition of partial products in each set CE_{ci} generates a sum bit and other carry bits which are propagated to partial products of the next column CE_{ci+1}. Because of these carry bits, elements of CE_{ci} cannot be summed independently without taking into consideration carry bits propagated from adding elements of the previous columns CE_{ci-i}, for $0<i<ci$. The summation between partial products in CE_{ci} and the propagated carry bits from previous column results the output bit z_{ci} and new carry bits to the next column $ci+1$. We denote the integer quantity of these carry bits as CO_{ci+1}, and the quantity of carry bits that propagate to the column ci as CO_{ci}, e.g., consider again Figure 13, CO_6 is an integer variable representing the value of carry bits that propagate to column 6, and CO_7 is the value of carry bits which are generated from column 6 and propagate to column 7. Based on this description, we formulate the addition tree equation of the column ci as

$$z_{ci}+2\cdot CO_{ci+1}=CO_{ci}+\sum_{ri=0}^{n-1}pp_{ci,ri}. \tag{1}$$

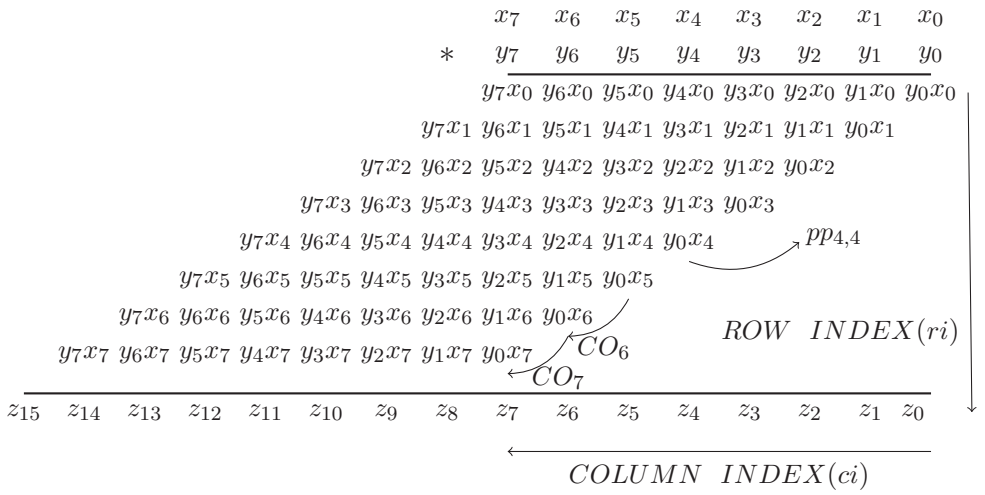

Figure 13: Addition Tree of 8-bit Multiplier

.

So far, we have presented some definitions of the multiplication function. In the following, we prove that not only the multiplication function is a p.r. function, but also its partial functions that generate and add partial products are also p.r. functions. This proof allows to verify the generation and addition of every partial product in a multiplier using the ECRC technique (see Section 3.1). Verifying a multiplier then boils down to check the whole addition tree. For this purpose, we introduce the *combined multiply-add* (CMA) function since it can define the generation and the addition of a partial product, and show that the CMA is a p.r. function, therefore, its implementation can be verified using the ECRC.

Definition 20 (Combined Multiply-Add (CMA))**.** *The combined multiply-add function* $CMA(x_0, x_1, Y) = x_0 \cdot x_1 + Y$ *combines the bitwise multiplication function and the addition function, where* Y *is an integer quantity,* x_0 *and* x_1 *are Boolean variables.*

Lemma 3. *The CMA function* $CMA(x_0, x_1, Y)$ *is a p.r. function.*

Proof. Based on Theorem 2, the CMA function is defined by the relations:

$$CMA(0, x_1, Y) = Y$$

$$CMA(x_0 + 1, x_1, Y) = Add(CMA(x_0, x_1, Y), x_1).$$

The first relation is a projection, while the addition function *Add* of the second relation is known to be a p.r. function. This implies that the CMA function is itself a p.r. function. ■

3.2.2 *Overview of the Approach*

As demonstrated in the previous subsection, a multiplier can be defined as an addition tree of partial products which are generated using bitwise multiplications. Based on that, we propose an approach that applies case splitting to check in each case the generation of only one partial product as well as the correctness of adding this partial product to other partial products of the multiplier. We will refer to this partial product as the *Partial Product Under Verification* (PPV). The combination of the generation and the addition of PPV can be defined using the CMA function from Definition 20. Since the CMA function is a p.r. function, its implementation can be checked using the ECRC.

The approach applies the ECRC on the cone that is influenced by generating and adding the PPV to other partial products. This *PPV cone* can be described using the addition tree and the addition tree equation which are defined in Subsection 3.2.1. As stated there, the PPV belongs to a set CE_{cv} of a column cv in the addition tree, and the function of the related output z_{cv} is formulated using the addition tree equation. Based on this description, we define the PPV cone as the logic gates that are in the functional support of z_{cv} which represent the generation and the addition of the following partial products:

1. Those that are in sets CE_{cv-i}, for $0 < i < cv$, which produce carry bits that is added to the PPV.

2. All partial products in the set CE_{cv} which includes the PPV itself since they are involved in the addition functions that produce together the output z_{cv}.

3. Partial products of the set CE_{cv+1} which receive carry bits that resulted from adding PPV to other partial products.

The approach decomposes the verification into n^2 cases, which corresponds to the number of partial products of n-bit multiplier. In every case, it extracts the PPV cone and applies equivalence checking based on the CMA recurrence relation. This case split leads to small differences between the compared implementations of the constructed miters, thereby the approach avoids redundant checks between the cases, which allows fast equivalence checking regardless of

the multiplier size. In the following, we will refer to this approach as the *Checking Partial Product* (CPP) approach.

3.2.3 *Mathematical Formulations*

The CPP approach aims in each splitting case to verify the combination of generating and adding the PPV which is a CMA function $x_0 \cdot x_1 + Y$. The approach verifies this function by extracting its implementation from the DUV (the PPV cone) together with checking the consistency of the cone with the CMA relations. Consider Equation (1) to formulate a mathematical expression for the PPV cone, PPV is a partial product, then it can be expressed as $pp_{cv,rv} = x_{rv} \cdot y_{cv-rv}$, where cv and rv are indices of PPV which range over the ranges of the column index ci and the row index ri. PPV belongs to a column in the addition tree whose output function z_{cv} is formulated using the addition tree equation

$$z_{cv} + 2 \cdot CO_{cv+1} = CO_{cv} + \sum_{\substack{ri=0 \\ ri \neq rv}}^{n-1} pp_{cv,ri} + pp_{cv,rv}. \tag{2}$$

As can be seen from Equation (2), adding PPV generates the carry CO_{cv+1} that propagates to the column $cv + 1$. Therefore, PPV addition has influence on the column $cv + 1$. The addition tree equation of this column is

$$z_{cv+1} + 2 \cdot CO_{cv+2} = CO_{cv+1} + \sum_{ri=0}^{n-1} pp_{cv+1,ri} \tag{3}$$

Because the structural relations between the addition tree equations are summations, to formulate the PPV cone, Equation (2) that formulates the cone of z_{cv} is added to Equation (3) that formulates the cone of z_{cv+1}. Note that column $cv + 1$ has higher weight with 2^1 than column cv, so that Equation (3) will be multiplied by 2, then it is summed to Equation (2), which derives the PPV cone equation

$$z_{cv} + 2 \cdot z_{cv+1} + 4 \cdot CO_{cv+2} = CO_{cv} + \sum_{\substack{ri=0 \\ ri \neq rv}}^{n-1} pp_{cv,ri} + \sum_{ri=0}^{n-1} 2 \cdot pp_{cv+1,ri} + pp_{cv,rv} \tag{4}$$

Note that the term $2 \cdot CO_{cv+1}$ is removed from the two sides of Equation (4). We refer to the integer quantity that are added to PPV as

$$Q_{rv} = CO_{cv} + \sum_{\substack{ri=0 \\ ri \neq rv}}^{n-1} pp_{cv,ri} + \sum_{ri=0}^{n-1} 2 \cdot pp_{cv+1,ri},$$

and we substitute the PPV term $pp_{cv,rv}$ by $x_{rv} \cdot y_{cv-rv}$, which reformulates Equation (4) to

$$z_{cv} + 2 \cdot z_{cv+1} + 4 \cdot CO_{cv+2} = x_{rv} \cdot y_{cv-rv} + Q_{rv} \tag{5}$$

The mathematical expression of the PPV cone is couched by the right side of Equation (5). Note that the PPV cone is a bitwise multiplication followed by addition, therefore, it is an implementation of the CMA function. By replacing the variable a_0 of the CMA recurrence relation with x_{rv}, the variable a_1 with y_{cv-rv}, and the variable B with Q_{rv}, we apply the CMA relations on the PPV cone in that Equation (5) becomes

$$z_{cv} + 2 \cdot z_{cv+1} + 4 \cdot CO_{cv+2} = CMA(x_{rv}, y_{cv-rv}, Q_{rv})$$

The initial relation of PPV is $CMA(0, y_{cv-rv}, Q_{rv}) = Q_{rv}$, by assigning zero to x_{rv}. Since the value of Q_{rv} depends on the value of other partial products of the multiplier, checking the initial relation for every partial product separately is not trivial. The approach overcomes this problem by checking the initial relations of all partial products together. Because for each initial relation $x_{rv} = 0$, checking all initial relations together is done by assigning zeros to all x_i bits, which implies that $X = \sum_{i=0}^{n-1} 2^i x_i = 0$. At $X = 0$, the approach checks trivially that the result of the DUV is equal to zero, where $0 \cdot Y = 0$. Thus, checking the initial CMA relations together is done by checking the initial relation of the multiplier.

After verifying the initial relation, it is the time to check the recurrence relation of PPV:

$$CMA(x_{rv} + 1, y_{cv-rv}, Q_{rv}) = CMA(x_{rv}, y_{cv-rv}, Q_{rv}) + y_{cv-rv}. \tag{6}$$

The approach checks this recurrence relation for every PPV cone using the ECRC. It builds a miter that compares an implementation for the left side of the recurrence relation against another implementation representing the right

side of this relation. The implementation details of the miter will be explained in the next subsection.

By checking the addition and the generation of all partial products in the multiplier, the approach either announces the consistency of the DUV with the multiplication function or determines the part of the DUV (the PPV cone) that has a bug.

3.2.4 *Implementation*

Implementing the recurrence relation of Equation (6) as it is, in every splitting case, causes redundant checks. The operation $x_{rv} + 1$ propagates a carry bit of value one to other bits of X which has higher weights, i.e., $x_{rv+1}, x_{rv+2}, \cdots, x_{n-1}$. As the value one will be added to these bits in other splitting cases, it is redundant to implement this operation $x_{rv} + 1$ as an addition. To simplify the implementation of the equivalence checking and remove these redundant checks, the approach assigns zero to the bit x_{rv}, at this case $x_{rv} + 1$ does not generate a carry bit. Therefore, the one bit adder $x_{rv} + 1$ in the left side of Equation (6) is implemented by the XOR function $x_{rv} \oplus 1$ which is the inversion of x_{rv} ($\bar{x}_{rv}$). Because of this optimization the carry bit that results from $x_{rv} + 1$ to higher bits are stopped, and building the equivalence checking miter becomes easier. This optimization has no influence on the soundness of the verification process since at $x_{rv} = 0$, still all patterns of the $pp_{cv,rv}$ are checked, whereas $pp_{cv,rv} = 1 \cdot y_{cv-rv}$ in the implementation of the left side of Equation (6) and $pp_{cv,rv} = 0 \cdot y_{cv-rv}$ in the implementation of the right side, thus such assigning will not affect the coverage of the approach.

We implemented our approach by integrating it with the tool ABC presented in [72]. The approach creates n^2 splitting cases. In every case, it builds a miter circuit based on Equation (6), checking the generation and the addition of one partial product PPV by performing the following steps:

1. It extracts the PPV cone by getting the gates that are connected to the output bits z_{cv} as well as z_{cv+1} since these gates represent the function $z_{cv} + 2 \cdot z_{cv+1}$ which results from adding the PPV to other partial products. This task is called partial products decomposition shown in the left side of Figure 14.

2. It searches the extracted cone for the gate that generates the PPV, in addition to that it makes two copies from the cone and assigns zero to the bit x_{rv} of the two copies. The variable x_2 in the example of Figure 15

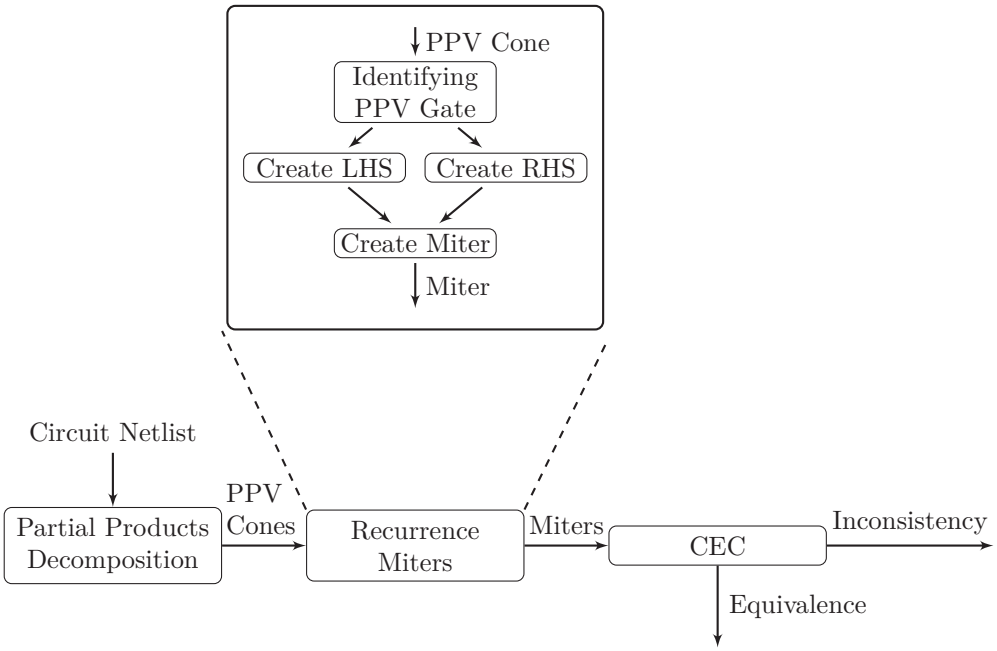

Figure 14: Detailed Flow of CPP Approach

(upper and lower part) is an instance of x_{rv}. We refer to this step as identifying PPV shown in the upper part of Figure 14.

3. The next step is creating the left side (LHS) of the miter, as demonstrated in Figure 14, it takes a copy of the PPV cone and inverts explicitly the bit x_{rv} of the identified PPV gate. An instance of such step is shown in the upper part of Figure 15, where it inverts the bit x_2 of the partial product $x_2 \cdot y_2$.

4. For creating the right side (RHS) which is demonstrated also by Figure 14, the outputs z_{cv} and z_{cv+1} of the second copy of the PPV cone is added to the input y_{cv-rv} using an external 2-bit adder. This can be seen in the lower part of Figure 15, where the input bit y_2 is added to output bits $z_4 + 2 \cdot z_5$.

5. The construction of a recurrence miter is completed, as shown in Figure 14, with the usual comparison logic by XOR-ing and OR-ing the two output bits of the left side with two output bits of the external adder in the right side.

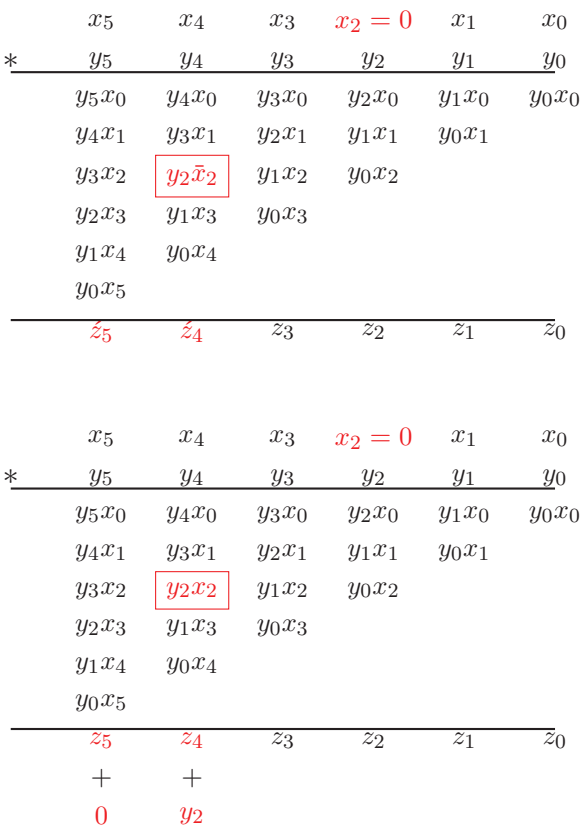

Figure 15: Inputs of Miter for Checking $pp_{4,2} = y_2 \cdot x_2$

Finally, as seen in the last block of Figure 14, the constructed miters are given independently to the Combinational Equivalence Checking (CEC) approach of the ABC which announces the equivalence or inconsistency between the compared sides of the miters.

In general, the CPP approach shows significant results in the verification of multipliers of various architectures as demonstrated by the experimental evaluation.

3.2.5 *Discussion*

Miters that are constructed based on the recurrence relation of Equation (6) offer large number of internal equivalences in that equivalence checking approaches like *rewriting* and *fraiging* [13] succeed to prove the equivalence (or non-equivalence) of many cases without resorting to the SAT solver. This ex-

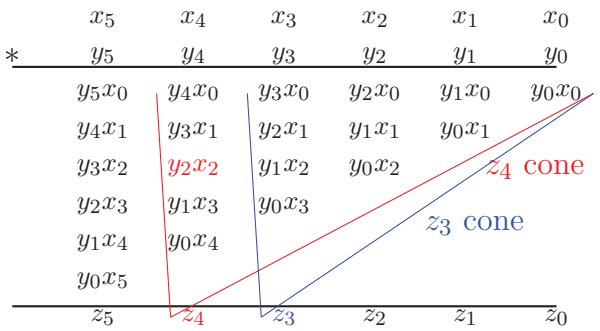

Figure 16: Search of CPP Approach for $pp_{4,2} = y_2 \cdot x_2$

plains the effectiveness of the CPP approach, even when it is applied to multipliers larger than 32 bits.

The reasons behind the extraordinary efficient of the constructed miter are:

1. The lower parts of both sides of the miter that produce the carry CO_{cv} have identical structural.

2. The difference between compared implementations is only in the generation of PPV and the propagation way of CO_{cv+1}. In the left side, CO_{cv+1} is added to the elements of CE_{cv+1} through the cone of PPV, while in the right side, CO_{cv+1} is separated into two quantities, one quantity is added using the logic gates of the PPV cone, whereas the second quantity is added through the external adder. This difference is not huge since it is restricted to the part of the PPV cone that adds CO_{cv+1} to the partial products of the set CE_{cv+1}.

A nice further feature provided by the CPP approach is the capability to identify the location of a fault if exist in the DUV. Once sides of a miter of a PPV cone are proved to be not equivalence, it is obvious to deduce that the faults are in the gates that belong to this PPV cone and are not in previous checked cones. This feature helps the debugging of incorrect multipliers because it determines a narrow boundary for gates that at least one of them has a fault.

Also, these constructed miters can be checked in parallel or serially since the cases are independent of each other. We use parallelism for multipliers larger than 32 bits.

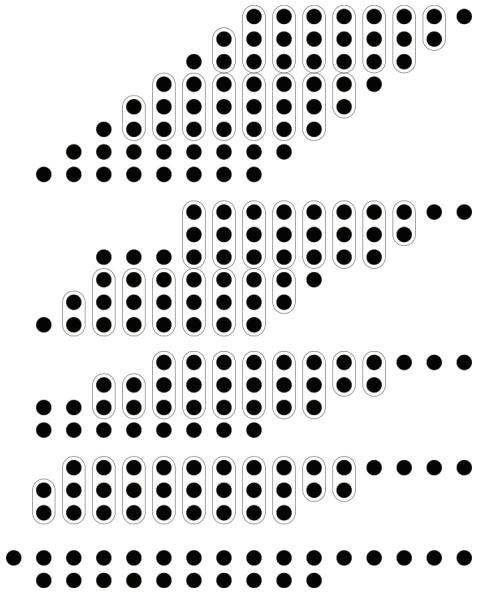

Figure 17: Wallace Tree Accumulation

3.2.6 *Limitations*

However, the approach suffers from two main limitations that restrict its applicability for specific architectures. The first limitation is related to the dependency between gates that generate partial products, while the second is because of an assumption about the propagation of carry bits within the multiplier. In the following, we explain in details these two limitations. The first one is called *partial products dependency*, and we refer to the second limitation as *carry propagation assumption*.

Partial Products Dependency Limitation

To clarify this limitation, consider Step 2 of the approach. The approach searches for the gate of the partial product $pp_{cv,rv}$. The search is done by comparing gates of cone z_{cv} and cone z_{cv-1}. The gate that belongs to the cone of z_{cv}, is not in the cone of z_{cv-1}, and has the input bit x_{rv}, is the one that the approach searches for. Figure 16 shows an instant of this search, where the gate that generates the partial product $PP_{4,2}$ is identified since it is in the cone of z_4 but not in z_3 cone and it gets the input variable x_2. After finding the gate of the PPV, the approach inverts the input bit x_{rv} of this gate. This inversion does not always affect on the partial product $pp_{cv,rv}$, but may affect other par-

tial products, which is the case for multipliers based on Booth recoding and optimized multipliers. Thus, if the implementation of partial products in a multiplier does not compute each product independently (as AND gate), the CPP approach fails to deal with such multiplier.

Carry Propagation Assumption Limitation

The second limitation is related to the assumption that carry bits generated because of adding $pp_{cv,rv}$ (CO_{cv+1}) propagate to other partial products of the multiplier through a specific path. In this path, these carry bits are fed as inputs in order to be added to the elements of next column (CE_{cv+1}) resulting in another group of carry bits (CO_{cv+2}). Because of this assumption, the approach checks only during each splitting case the correctness of the propagation of CO_{cv+1} bits to CE_{cv+1} elements. It bypasses the check of the propagation of carry bits of CO_{cv+1+i} through elements in the sets CE_{cv+1+i}, for all $0 < i \leq 2n - cv - 2$, since these propagations will be tested during next splitting cases.

This assumption is valid for classical architectures of multipliers such as those that are based on Wallace tree or Dadda tree. As shown in Figure 17, Wallace tree accumulates partial products of the addition tree into two arrays of bits where partial products and resulted bits are symbolized as dots. The accumulation is performed by adding each two or three elements of a column, where each addition operation results in two bits: a sum bit which replaces those added elements in the column and a carry bit which propagates to the next column. This implies that all carry bits generated because of adding elements of a column are fed to the next column. Therefore, carry bits in the Wallace tree propagate through paths that are consistent with the carry propagation assumption.

However, the assumption is not valid for multiplier architectures where some carry bits of CO_{cv+1} propagate without adding them to elements of CE_{cv+1}. In this case, the correctness of the generation and the propagation of these carry bits will not be checked by the CPP approach. Because of that: 1) the approach cannot verify architectures that do not satisfy the carry propagation assumption, and 2) it can only find faults of a certain class which excludes those faults that invalidate this assumption.

3.3 EXPERIMENTAL RESULTS

Our approach is built on top of the ABC tool [13]. ABC compiles the miter circuit into an And-Inverter Graph (AIG) and applies structural reduction techniques like fraiging and rewriting. These techniques reduce effectively the size

Table 2: Runtimes for Verification of Multipliers

Benchmark	I/O bits	CPP [h:m:s]	Fujita [h:m:s]
SP-AR-RC	16/32	00:01:23	00:00:31
SP-WT-CL	16/32	00:00:46	00:09:08
SP-WT-BK	16/32	00:00:52	00:10:05
SP-CT-BK	16/32	00:00:43	00:17:56
SP-AR-RC	32/64	02:34:40	11:09:18
SP-WT-CL	32/64	00:15:12	TO
SP-DT-HC	32/64	00:21:14	TO
SP-CT-BK	32/64	00:21:20	TO
SP-AR-RC	48/96	20:32:12	TO
SP-WT-CL	48/96	01:29:36	TO
SP-DT-HC	48/69	03:53:17	TO
SP-CT-BK	48/96	01:20:00	TO
SP-AR-RC	64/128	94:37:20	TO
SP-WT-CL	64/128	05:46:40	TO
SP-CT-BK	64/128	05:31:44	TO
SP-AR-RC	128/255	TO	TO
SP-CT-BK	128/255	78:11:12	TO

of the AIG if the miter has many similar internal nodes. At the backend of the equivalence checking procedure, the reduced AIG is converted to Conjunctive Normal Form (CNF) and the resulting instance is given to MiniSAT [37].

All experiments have been carried out on an Intel(R) Core(TM) i5-3320M CPU (2.6 GHz, 16 GByte) running Linux. For the experiments, the multipliers are given as Verilog RTL code. The synthesis of the designs to a gate level netlists has been done using the *Yosys Open Synthesis Suite* [101] and ABC.

We conducted two types of experiments. The first one demonstrates the practical time of the approach in checking larger multipliers, while the second experiment is to show the capability of discovering bugs.

3.3.1 *Equivalence Checking Results*

We compare the runtimes of our CPP approach against Fujita's approach. In the original approach of Fujita the equivalence is checked using BDDs. Here,—and for a fair comparison—we use Fujita's approach with ABC as a backend for equivalence checking.

The first column of Table 2 shows the name of the circuit. The second column gives the number of inputs and output bits. The next two columns provide the runtimes. Note that the runtimes of Fujita's approach include only the verification of the lower n bits (not the full $2n$ output bits). The time out (TO in the table) has been set to 100 hours. Please note that for a naive miter construction (one big miter) and then running the ABC command (CEC) *all* benchmarks timed out after 100 hours.

The experiments show that the verification time of the CPP approach depends not only on the size of the multiplier circuit, but also on the type of the partial products accumulator. The circuits with Wallace tree or (4,2) compressor are verified in less time than those with array accumulator. As can be seen our approach verifies the correctness of the multipliers for up to 128 bits in practical time. Fujita's approach fails here already for the complex architectures.

3.3.2 *Fault Injection*

In order to demonstrate the ability to discover bugs, we applied our approach to faulty designs that have been created by automatic fault injection. The faults are inverters that have been injected in the AIG representation of the netlist. We applied the CPP approach on different copies of each netlist where each copy contains one single fault. The approach has succeeded to discover all injected bugs. The results are summarized in Table 3. The first two columns describes the type of the multiplier architecture (as explained in the previous section) and the bit width, respectively. The third column gives the size (number of nodes) of the AIG. The number of performed runs is given in the fourth column, and the average run-time needed to discover the bug is given in the last column.

For all 16 bit architectures, we systematically covered the whole AIG by injecting two faults for each node. For the larger designs, random gates were chosen with an even distribution over the netlist. For the 32 bit multipliers it can be observed that the runtimes vary between fractions of a second and

Table 3: Results for Models with Injected Faults

Benchmark	I/O bits	|AIG|	#Faults	∅ runtime
SP-AR-RC	16/32	2126	4252	2.56 *s*
SP-WT-CL	16/32	2988	5976	1.80 *s*
SP-CT-BK	16/32	2201	4402	0.45 *s*
SP-AR-RC	32/64	14196	710	27.05 *s*
SP-WT-CL	32/64	12741	319	59.72 *s*
SP-CT-BK	32/64	9173	459	10.00 *s*

several minutes, where usually more than half of the bugs are discovered in less than a second.

To summarize, the results show that our approach works well for bug-hunting as well as for the verification of correct multiplier designs.

3.4 SUMMARY AND FUTURE WORK

Verification of bit-level multipliers still has no general automated solution. In this chapter, we verify multipliers at the gate-level using the ECRC technique which does not require neither information about high-level designs of multipliers nor a golden reference for the comparison. We have developed an approach that allows to verify multiplier circuits up to 128 bits. The approach is based on functional properties of the multiplication function which can be expressed as recurrence equation, together with a new case splitting scheme. As a consequence enough similarities remain for the equivalence check of each case. Overall, the approach increases the scalability of equivalence checking to verify larger multipliers, however, it cannot be applied for all types of multiplier architectures such as Booth recoding multipliers. Also, it can detect faults that belong only to a certain class, whereas faults that break an assumption about the structural of the multiplier may not be discovered by the approach.

In future work, an investigation should be conducted to overcome the limitation of the CPP approach by rewriting the gate netlists of multipliers to isolate partial products that rely on each other. Furthermore, the ECRC technique could be leveraged for the verification of other functions that can be defined by recurrence relations such as Fused Multiply Add ($Xo = X_1 \cdot X_2 + X_3$), where X_1, X_2 and X_3 are integer inputs, while Xo is the integer output of the function.

4

SYMBOLIC COMPUTATION FOR VERIFYING COMPLEX MULTIPLIERS

Formal verification utilizing symbolic computation has demonstrated the ability to formally verify large *Galois field* arithmetic circuits [68] and basic architectures of integer arithmetic circuits [23, 39]. The technique models the circuit as Gröbner basis polynomials and reduces the polynomial equation of the circuit specification wrt. the polynomials model using the *Ideal Membership Testing* (IMT). However, during the polynomials reduction by the IMT, the technique suffers from an exponential blow-up of the size of the polynomials, in particular, when it verifies *Parallel Prefix Adders* (PPAs) and *Booth recoded* multipliers. In this chapter, we analyze the computational complexity of verifying integer multipliers by the symbolic computation technique. Moreover, we address the reasons of the exponential blow-up that occurs with complex integer multipliers and introduce an algorithm that allows to apply the technique to a large class of multiplier circuits, i.e., including basic and parallel multiplier architectures. The goal of this algorithm is rewriting the Gröbner basis model of a multiplier circuit in order to replace its bit-level description by a network of adder cells. Based on our observation and previous observations [22, 60, 96], the verification problem is polynomially bounded in both space and time when it is applied to multipliers described as *full adder networks*. To circumvent the exponential complexity of verifying bit-level multipliers, we propose the *model rewriting* algorithm to convert any kind of multiplier architectures into what we call *Sum Carry Networks* (SCNs) which are networks of adder cells connected by sum and carry signals. In contrast to full adder networks that are limited to half and full adders, SCNs consist of adders with arbitrary number of input bits. Since the reduction complexity of SCNs as well as full adder networks wrt. their specification polynomials by the IMT procedure is polynomial, rewriting bit-level multiplier models into SCNs before starting the reduction process by the IMT circumvents the exponential blow-up of the number of terms during this reduction process. To perform the rewriting step, as shown in Figure 18, the proposed model rewriting algorithm executes successively two novel schemes named *XOR rewriting* and *common rewriting* on models of bit-level multipliers and applies a logic reduction rule within the XOR rewriting scheme. The algorithm lifts bit-level models into SCNs, concurrently, it identifies and then

removes by a logic reduction rule redundant terms that evaluate to zero. These terms are distributed within models of complex multiplier architectures. Without early vanishing to such terms, their number increases exponentially during the reduction process, making the verification task computationally infeasible. Besides introducing the model rewriting algorithm, this chapter presents *substitution rules* that enhance the computational performance of the IMT (shown on the right side of Figure 18) to verify large scale complex multipliers. As observed by [23, 39], a proper substitution order is crucial to circumvent the blow-up within the IMT. Restricting the IMT to follow a fixed substitution order (see Subsection 2.2.3) makes the technique applicable for a specific multiplier architecture. The proposed substitution rules qualify the IMT to find the proper substitution order that improves the time performance of the IMT as well as expands its applicability. The experiments show that the enhanced symbolic computation technique is applicable to verify a large class of multiplier circuits of up to 128-bit. These main contributions and other contributions in this chapter can be summarized as follows:

1. Comparing models of integer multipliers over the Boolean ring (utilized in thesis) versus *binary Galois field* models with respect to the complexity of the verification problem.

2. Observing and justifying that the verification complexity of integer multipliers expressed as SCNs is polynomial in space.

3. Determining the reason for the inefficiency of applying the symbolic computation technique to verify complex multipliers consisting of Booth partial products and PPAs.

4. Observing that rewriting as an explicit step before calling the IMT procedure is capable of circumventing blow-ups during reduction of polynomials since it rewrites models of complex multiplier architectures into SCNs. For this propose, rewriting schemes and a logic reduction rule have been proposed to remove *vanishing monomials* (monomials that always evaluate to zero) that appear in the algebraic models of complex integer multipliers.

5. Supporting the IMT procedure by substitution rules that permit to find a proper substitution order regardless of the multiplier architecture.

6. Adding modulo 2^{2n} to the specification of n-bit integer multipliers, which is crucial to match the specification with multipliers that consist of Booth partial products or redundant binary addition trees.

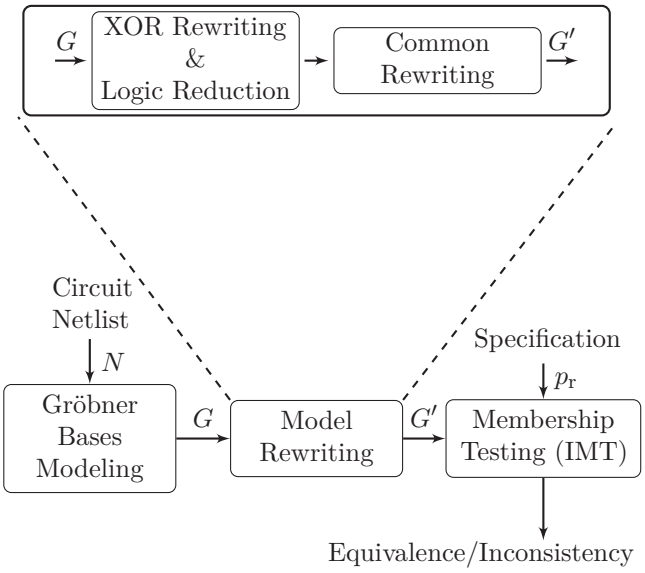

Figure 18: Enhanced Symbolic Computation for Verifying Multipliers

4.1 BOOLEAN RING VERSUS BINARY GALOIS FIELD

In the literature, two methods have been proposed to model bit-level circuits as Gröbner bases. The first is modeling the circuit by polynomials in the Boolean ring ($\mathbb{Z}_2$) as in the thesis, while the second method models the circuit in the binary Galois field ($\mathbb{GF}_2$) as in [14, 68, 78]. What distinguishes $\mathbb{GF}_2$ from the ring $\mathbb{Z}_2$ is that all the field operations are performed modulo an irreducible primitive polynomial and the coefficients are reduced modulo two. This implies that all coefficients over $\mathbb{GF}_2$ take a value from the set $\{0, 1\}$, whereas $-1 = +1$ and $2 \mod 2 = 0$. Every Boolean logic gate in the circuit is mapped to a polynomial function over $\mathbb{GF}_2$ as follows:

NOT:	$x_o = \neg x_1$	$\Longrightarrow$	$x_o + x_1 + 1 \mod 2$
AND:	$x_o = x_1 \wedge x_2$	$\Longrightarrow$	$x_o + x_1 x_2 \mod 2$
OR:	$x_o = x_1 \vee x_2$	$\Longrightarrow$	$x_o + x_1 x_2 + x_1 + x_2 \mod 2$
XOR:	$x_o = x_1 \oplus x_2$	$\Longrightarrow$	$x_o + x_1 + x_2 \mod 2$
MUX:	$x_o = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3)$	$\Longrightarrow$	$x_o + x_1 x_2 + x_1 x_3 + x_3 \mod 2.$

In this section, we discuss an argument claiming that modeling integer multiplier circuits over $\mathbb{GF}_2$ reduces the computation complexity of the verification problem compared to the modeling over $\mathbb{Z}_2$. What supports this argument is that models of integer circuits over $\mathbb{GF}_2$ are free from nonlinear terms that their coefficients are multiples of two which appear intensively in $\mathbb{Z}_2$ models, e.g., the

XOR gate is modeled over $\mathbb{GF}_2$ as $x_o = x_1 + x_2 \mod 2$ without the nonlinear term $(-2x_2x_1)$ that appears in the polynomial of the XOR function over $\mathbb{Z}_2$ which is $x_o = -2x_2x_1 + x_1 + x_2$. Hence the argument claims that verifying $\mathbb{GF}_2$ models has less computationally cost. In the following, we show that this argument is not true, on the contrary, the computational complexity of verifying integer multipliers modeled over $\mathbb{GF}_2$ is exponential, while this verification problem turns out to be tractable by leveraging the algorithms proposed in this chapter which are performed on $\mathbb{Z}_2$ and cannot be applied to $\mathbb{GF}_2$ models.

The significant capability offered by $\mathbb{Z}_2$ and is missed in $\mathbb{GF}_2$ is the applicability to check together the correctness of Boolean functions that depend accumulatively on each other—they can be encoded to one integer function. Integer adder and multiplier circuits are implemented as a tuple $F = (f_0, \cdots, f_m)$ of multi-output Boolean functions, whereas f_i is an one-output Boolean function for each $i \in \{0, \cdots, m\}$ and F is an integer-valued function related to the functions of its tuple by the equation $F = \sum_{i=0}^{m} 2^i \cdot f_i$. Sophisticated algorithms such as those proposed in the thesis take this relationship into consideration to reduce the complexity of the verification problem of integer circuits to be solvable. In fact, this relationship can be considered if the output Boolean functions are in the ring $\mathbb{Z}_2$, while it is ignored if the functions are modeled over $\mathbb{GF}_2$.

Lemma 4. *For an integer-valued function F implemented as a tuple of one-output Boolean functions $F = (f_0, \cdots, f_m)$, the relationship that relates these Boolean functions $F = \sum_{i=0}^{m} 2^i \cdot f_i$ cannot be considered during their verification process over $\mathbb{GF}_2$.*

Proof. Since the Boolean functions f_i are modeled over $\mathbb{GF}_2$, their relationship $F = \sum_{i=0}^{m} 2^i \cdot f_i$ must also be modeled over $\mathbb{GF}_2$ to be considered in the verification process, otherwise, the process is not sound. In $\mathbb{GF}_2$, coefficients are reduced modulo two, therefore, $(F = \sum_{i=0}^{m} 2^i \cdot f_i) \mod 2$ is rewritten to $F \mod 2 = f_0$, resulting in an equation included only one output Boolean function f_0, which proves that the relationship among outputs of an integer function cannot be utilized in the $\mathbb{GF}_2$ field. ■

Because of Lemma 4, output Boolean functions modeled over $\mathbb{GF}_2$ can only be checked by considering them independently from each others. To illustrate this concept, consider the example of a half adder circuit.

Example 9. *A 2-bit adder (half adder) is implemented by two Boolean functions: one for the sum* $s = x_1 \oplus x_2$ *and another for the carry-output* $co = x_1 \wedge x_2$*, where* x_1 *and* x_2 *are Boolean inputs.*

Over $\mathbb{Z}_2$*, the 2-bit adder is modeled by two polynomials:*

$$g_1 := -s - 2x_1x_2 + x_1 + x_2 \qquad g_2 := -co + x_1x_2.$$

The output functions s *and* co *can be encoded to the integer function* $F = 2co + s = x_1 + x_2$ *which derives the functional specification polynomial* $p_{\mathrm{r}} := -2co - s + x_1 + x_2$*. To verify the 2-bit adder, the IMT is evoked to test whether the polynomials set* $\{g_1, g_2\}$ *satisfies the specification* p_{r}*. This is performed by substituting iteratively* co *and* s *in* p_{r}*. First,* co *is replaced by* x_1x_2 *which is the tail term of* g_2 *based on the rewrite rule (see Definition 17). The result of this substitution (division) is the remainder* $r := -2x_1x_2 - s + x_1 + x_2$*. Second,* s *is substituted for the tail terms of* g_1 $(-2x_1x_2 + x_1 + x_2)$*, resulting in a new remainder* $r := -2x_1x_2 - 1 \cdot (-2x_1x_2 + x_1 + x_2) + x_1 + x_2 = -2x_1x_2 + 2x_1x_2 - x_1 - x_2 + x_1 + x_2 = 0$*. All terms of the final remainder cancel each other and the remainder is equal to zero, hence an equivalence is announced between the 2-bit adder and its functional specification.*

In the case of $\mathbb{GF}_2$*, the 2-bit adder is modeled by the polynomials:*

$$\hat{g}_1 := s + x_1 + x_2 \mod 2 \qquad \hat{g}_2 := co + x_1x_2 \mod 2.$$

As discussed before, it is not possible to describe the functional specification by one polynomial that relates the two outputs s *and* co*. The functional specification is modeled by two polynomials: one to validate the sum which is* $\hat{p}_{\mathrm{r}_1} := s + x_1 + x_2 \mod 2$ *and another for the carry-output* $\hat{p}_{\mathrm{r}_2} := co + x_1x_2 \mod 2$*. The IMT is invoked two times to compare* $\hat{p}_{\mathrm{r}_1}$ *against* $\hat{g}_1$ *and* $\hat{p}_{\mathrm{r}_2}$ *against* $\hat{g}_2$*. It can be easily seen that the IMT returns zero remainders at the two cases, which proves the correctness of the 2-bit adder circuit.*

Yet, we have demonstrated that output Boolean functions of an integer function must be checked independently, as long as they are modeled in the field $\mathbb{GF}_2$. But why this is a problem for the verification process. In the following, we answer this question by proving that the computational complexity of verifying these Boolean functions individually is exponential.

A Boolean function f_i can be linearly verified by the IMT procedure, as long as it is described by a canonical representation, since the verification process will turn out into a subtraction of two canonical representations for f_i and its specification. A $\mathbb{GF}_2$ model of f_i is canonical, if it is represented by one polynomial mapping primary outputs directly to primary inputs of f_i without any kind of internal variables, hence the complexity of verifying a function f_i by the IMT can be measured by the size of the polynomial which models canonically

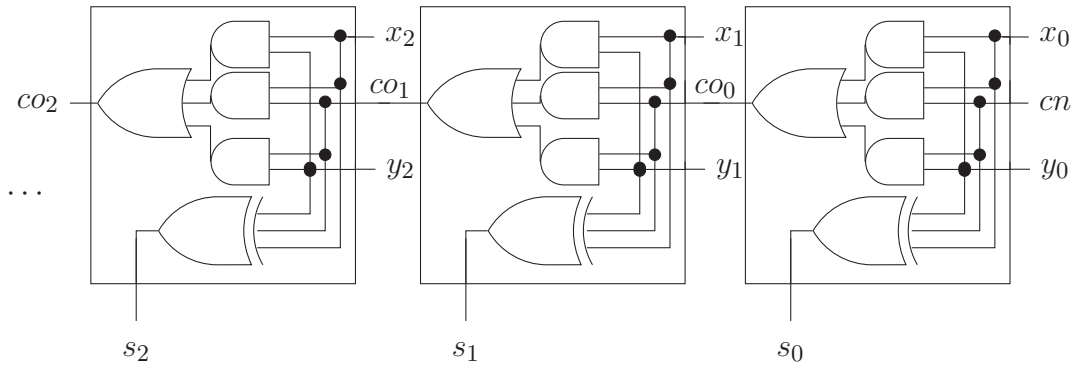

Figure 19: Array of Full Adders

this function. To estimate the verification complexity of integer adders and multipliers, we address first the canonical representations for arrays of 3-bit adders (full adders) since they are compounded to build what is called array adders and array multipliers (see Subsection 2.3.1). We exploit the symmetry of such array architectures in order to derive simple proofs about addition and multiplication functions, which can be generalized afterwards for other architectures.

Consider the array of adders cells shown in the Figure 19. Every 3-bit adder in the array includes 3-inputs OR function, the output of every OR is fed into two AND functions which their outputs are given as inputs to another OR, building a chain of OR and AND functions. The chain generates internal carry bits as outputs, every internal carry co_{i-1} is given as an input to the output Boolean functions s_i and to the function of the next internal carry co_i. Thus describing s_i by one canonical polynomial over $\mathbb{GF}_2$ is performed by eliminating successive internal carry variables co_j, for all $j \leq i-1$. In the following, we prove that because of these eliminations, for an array of $n+1$ adder cells, the size of the canonical polynomial of the output Boolean function s_n is $\mathcal{O}(2^{n+1})$, hence the complexity of verifying the array is exponential.

Lemma 5. *For an integer function F of a 3-bit adders array implemented as a tuple of Boolean functions $F = (s_0, \cdots, s_n)$ with a set of primary inputs $\{x_0, \cdots, x_n, y_0, \cdots, y_n, cn\}$ and modeled over $\mathbb{GF}_2$, the complexity to verify the output Boolean function s_n is bounded in space by $\mathcal{O}(2^{n+1})$.*

Proof. The integer function F is mapped in $\mathbb{GF}_2$ by the set of polynomials:
$co_j = (x_j \wedge y_j) \vee (x_j \wedge co_{j-1}) \vee (y_j \wedge co_{j-1}) \quad \Longrightarrow$
$g_{2j+1} := co_j + co_{j-1}y_i + co_{j-1}x_i + y_j x_j \mod 2$
$s_j = x_j \oplus y_j \oplus co_{j-1} \quad \Longrightarrow g_{2j} := s_j + co_{j-1} + y_j + x_j \mod 2$

$co_0 = (x_0 \wedge y_0) \vee (x_0 \wedge cn) \vee (y_0 \wedge cn) \implies$
$g_1 := co_0 + cn\, y_0 + cn\, x_0 + y_0 x_0 \mod 2$
$s_0 = x_0 \oplus y_0 \oplus cn \implies g_0 := s_0 + cn + y_0 + x_0 \mod 2,$
where $0 < j \leq n$ and the set of variables $\{co_0, \cdots, co_n\}$ model internal carry bits. To represent the function s_n canonically, a polynomial mapping s_n to the primary inputs of F is required. This is performed by backward substitutions using the rewrite rule in the polynomial that its leading variable is s_n, which is $(g_{2n} := s_n + co_{n-1} + y_n + x_n \mod 2)$. The first substitution replaces the variable co_{n-1} in g_{2n} with the tail terms of the polynomial $g_{2n-1} := co_{n-1} + co_{n-2}y_{n-1} + co_{n-2}x_{n-1} + y_{n-1}x_{n-1} \mod 2$, denoted as

$$g_{2n} \xrightarrow{g_{2n-1}} r_1 := s_n + co_{n-2}y_{n-1} + co_{n-2}x_{n-1} + y_{n-1}x_{n-1} + y_n + x_n \mod 2,$$

where r_1 is the resulted remainder polynomial. The next substitution

$$r_1 \xrightarrow{g_{2n-3}} r_2 := s_n + co_{n-3}y_{n-2}y_{n-1} + co_{n-3}x_{n-2}y_{n-1} + y_{n-3}x_{n-2}y_{n-1} + co_{n-3}y_{n-2}x_{n-1} + co_{n-3}x_{n-2}x_{n-1} + y_{n-3}x_{n-2}x_{n-1} + y_{n-1}x_{n-1} + y_n + x_n \mod 2$$

replaces another carry variable co_{n-2} in r_1 with the tail terms of polynomial $g_{2n-3} := co_{n-2} + co_{n-3}y_{n-2} + co_{n-3}x_{n-2} + y_{n-2}x_{n-2} \mod 2$ to obtain a new remainder polynomial r_2. These iterative substitutions are performed until the final remainder depends only on primary inputs of F, in other words until all successive internal carry variables co are substituted. Since the number of internal carry variables is n, the same number of iterative substitutions are executed which are denoted as $g_{2n} \xrightarrow{g_{2n-1}} r_1 \xrightarrow{g_{2n-3}} r_2 \xrightarrow{g_{2n-5}} r_3 \xrightarrow{g_{2n-7}} \cdots r_{n-2} \xrightarrow{g_3} r_{n-1} \xrightarrow{g_1} r_n$.

To calculate the size of the final remainder r_n, we prove that the number of terms of a remainder r_i ($|r_i|$) is equal to $2^i + |r_{i-1}|$, where $1 < i \leq n$, and r_i is obtained after the substitution iteration of index i in the previous remainder r_{i-1}. Each iteration i eliminates an internal carry variable co_{n-i} from the terms of polynomial r_{i-1}, substituting the variable co_{n-i} for three terms $(co_{n-i-1}y_{n-i} + co_{n-i-1}x_{n-i} + y_{n-i}x_{n-i})$ in order to get the next remainder r_i. This means that the size $|r_i|$ is larger than $|r_{i-1}|$ by double the number of terms that include co_{n-i}, which can be formulated as $|r_i| = 2 \cdot st_{i-1} + |r_{i-1}|$, where st_{i-1} is the number of terms incorporating the variable co_{n-i}. Since two of the three terms that have replaced co_{n-i} include another carry variable co_{n-i-1}, st_i (number of terms that include co_{n-i-1} in r_i) can be calculated as two times the number of terms that contain the variable co_{n-i} in the remainder

r_{i-1} (st_{i-1}). Based on this observation, another relation is deduced which is $st_i = 2 \cdot st_{i-1} = 2^2 \cdot st_{i-2} = 2^i \cdot st_1$. The first substitution started by the polynomial g_{2n} that has only one term of co_{n-1}, therefore, $st_1 = 1$, hence $st_i = 2^i \cdot st_1 = 2^i$. Consequently,

$$|r_i| = 2 \cdot st_{i-1} + |r_{i-1}| = 2^i + |r_{i-1}|.$$

Based on this derived equation,

$|r_n| = 2^n + |r_{n-1}| = 2^n + 2^{n-1} + |r_{n-2}| = 2^n + 2^{n-1} + 2^{n-2} + \cdots 2^2 + |r_1|$, as $|r_1| = 6$, then $|r_n| = 2^n + 2^{n-1} + 2^{n-2} + \cdots 2^2 + 2 + 1 + 3 = 2^n + 2^n - 1 + 3 = 2^{n+1} + 2$.

Since the size $|r_n|$ is equal to $2^{n+1} + 2$, the complexity to obtain a canonical representation for the function s_n is bounded in space by $\mathcal{O}(2^{n+1})$. Therefore, the computational complexity to verify the Boolean function s_n is also bounded by $\mathcal{O}(2^{n+1})$. This completes the proof.

■

Based on Lemma 4 and Lemma 5, the complexity of verifying an array of 3-bit adders of $n+1$ output bits modeled over $\mathbb{GF}_2$ is bounded by $\mathcal{O}(2^{n+1})$. Independently of the architecture of the integer adder, its canonical representation over $\mathbb{GF}_2$ is the same as an array of 3-bit adders since both of them have the same function, therefore, an integer adder is bounded by the same exponential verification complexity as an array of 3-bit adders. In the case of an integer multiplier, the architecture of the multiplier can be thought of as successive arrays of 3-bit adders to compress n arrays of partial products into one array, where the longest 3-bit adders array (last stage adder) has $2n-1$ outputs. Because the complexity of verifying the longest array is exponential in space by $\mathcal{O}(2^{2n})$, it is obvious to justify that the verification complexity of the array multiplier modeled over $\mathbb{GF}_2$ is also exponential. Based on this inference, it can be deduced that the verification of the multiplier over $\mathbb{GF}_2$ is exponential regardless of the multiplier architecture since all architectures represent the same function and therefore have the same canonical representation with an exponential size.

In the following sections, we show that, in contrast to $\mathbb{GF}_2$, the verification complexity of the multiplier is turned out to be polynomial by sophisticated algorithms executed over the Boolean ring models of multipliers.

4.2 VERIFICATION COMPLEXITY OF SUM CARRY NETWORKS

Verification of integer multipliers at gate-level suffers from exponential complexity in space and time, in particular, if the verification is performed by classical solvers such as SAT and BDDs, or if the multipliers are modeled in $\mathbb{GF}_2$ as has been demonstrated in the previous section. However, previous observations [22, 60, 96] have shown that the verification problem is polynomially bounded in both space and time, as long as it is applied to multipliers described as a network of full adders. Because of this observation, we have proposed a rewriting algorithm that lifts implicitly bit-level descriptions of a large class of multiplier architectures into networks of adder cells which are named Sum Carry Networks (SCNs), where the cells have arbitrary number of input bits. However, before introducing this rewriting algorithm, we are interested in analyzing the complexity of the IMT procedure for verifying SCNs modeled over the Boolean ring.

Definition 21 (Sum Carry Networks)**.** *Boolean ring models that map outputs $f_i(X)$ of integer-valued functions $F = (f_0, \cdots, f_n)$ to their inputs set $X = \{x_0, x_2, \cdots, x_{n-1}\}$ through adder cells functions $Fa = (s, co_0, \cdots, co_m)$ are named sum carry networks (SCNs). s and co are Boolean variables that denote sum and carry outputs of adder cells, respectively.*

The significant feature of SCNs is circumventing the exponential blow-up of the number of nonlinear terms that are distributed within models of integer-valued functions F. These nonlinear terms model the propagation of carry bits within F, in the thesis we name them *carry terms*. By describing these functions F as networks of adder cells, these carry terms are revealed only within polynomials of adder cells, which permits to cancel these terms simply before an exponential blow-up of their sizes. The adder cells of SCNs are not restricted to a specific size, e.g., a full adder, they have arbitrary number of input bits, which allows rewriting a large class of bit-level multiplier architectures into SCNs.

Definition 22 (Carry Terms)**.** *Carry terms are those nonlinear terms that model internal carry bits propagated among Boolean output functions of adder cells.*

According to our observation, these carry bits are modeled as nonlinear terms distributed among polynomials of Boolean ring models of adder cells, which are revealed, as long as each output function of Fa is expressed by one polynomial. These carry terms share monomials, while their coefficients have opposite signs and they are multiples of each other.

Example 10. *Consider the model of a 3-bit adder cell (full adder), which is described by one polynomial maps the output sum s to the primary inputs of the 3-bit adder $\{x_1, x_2, x_3\}$ and another polynomial models the function of the output carry co. Such model consists of the following polynomials:*
$g_1 : -co\ \boxed{-2x_3x_2x_1 + x_3x_2 + x_3x_1 + x_2x_1}$
$g_0 : -s\ \boxed{+4x_3x_2x_1 - 2x_3x_2 - 2x_3x_1 - 2x_2x_1} + x_3 + x_2 + x_1\ .$
Between these two polynomials carry terms are revealed—those that are colored blue, they share the monomials of the set $\{x_3x_2x_1, x_3x_2, x_3x_1, x_2x_1\}$, and their coefficients are multiple of each other and with different signs.

Yet, we have introduced the concept of SCN and its feature of revealing carry terms within polynomials of adder cells. To address the advantages of SCNs in reducing the verification complexity of integer functions, we start by analyzing the complexity of verifying a 3-bit adders array shown in Figure 19, when it is modeled as a SCN. This facilitates afterwards to justify that the verification complexity of n-bit integer multiplication function implemented as a SCN is bounded in space by $\mathcal{O}(n^2)$.

Lemma 6. *For a function F of a 3-bit adders array implemented as a tuple of Boolean functions $F = (s_0, \cdots, s_n)$ with a set of primary inputs $\{x_0, \cdots, x_n, y_0, \cdots, y_n, cn\}$ and modeled as a SCN, let s_i and co_i be a pair of Boolean variables representing the sum and carry-output of a 3-bit adder in the array F, where $0 \leq i \leq n$. The complexity to verify the SCN model of F wrt. its polynomial specification $p_{\mathrm{r}} := -2^{n+1}co_n - \sum_{k=0}^{n} 2^k s_k + \sum_{k=0}^{n} 2^k x_k + \sum_{k=0}^{n} 2^k y_k + cn$ is bounded in space by $\mathcal{O}(n)$ if the variables of each pair (co_i, s_i) have been substituted consecutively.*

Proof. The integer function F is modeled as a SCN by the set of polynomials:
$g_{2j+1} := -co_j\ \boxed{-2co_{j-1}y_jx_j + co_{j-1}y_j + co_{j-1}x_j + y_jx_j}$
$g_{2j} := -s_j\ \boxed{+4co_{j-1}y_jx_j - 2co_{j-1}y_j - 2co_{j-1}x_j - 2y_jx_j} + co_{j-1} + y_j + x_j$
$g_1 := -co_0\ \boxed{-2cn\,y_0x_0 + cn\,y_0 + cn\,x_0 + y_0x_0}$
$g_0 := -s_0\ \boxed{+4cn\,y_0x_0 - 2cn\,y_0 - 2cn\,x_0 - 2y_0x_0} + cn + y_0 + x_0,$
for $0 < j \leq n$. The verification process is performed using the IMT procedure by backward substitutions in the specification polynomial p_{r}. The process starts by substituting the variable co_n for the tail terms of the polynomial $g_{2n+1} := -co_n\ \boxed{-2co_{n-1}y_nx_n + co_{n-1}y_n + co_{n-1}x_n + y_nx_n}$, denoted as

$$p_{\mathrm{r}} \xrightarrow{g_{2n+1}} r_1 := -2^n s_n\ \boxed{+2^{n+2}co_{n-1}y_nx_n - 2^{n+1}co_{n-1}y_n - 2^{n+1}co_{n-1}x_n -}$$
$$\boxed{2^{n+1}y_nx_n} - \sum_{k=0}^{n-1} 2^k s_k + \sum_{k=0}^{n-1} 2^k x_k + \sum_{k=0}^{n-1} 2^k y_k + cn + 2^n y_n + 2^n x_n,$$

where r_1 is the resulted remainder. In the next substitution

$$r_1 \xrightarrow{g_{2n}} r_2 := -2^n co_n - \sum_{k=0}^{n-1} 2^k s_k + \sum_{k=0}^{n-1} 2^k x_k + \sum_{k=0}^{n-1} 2^k y_k + cn,$$

s_n in r_1 is replaced by the tail terms of polynomial $g_{2n} := -s_n$ $\boxed{+4co_{n-1}y_nx_n - 2co_{n-1}y_n - 2co_{n-1}x_n - 2y_nx_n} + co_{n-1} + y_n + x_n$, ensuing a new polynomial r_2 where the carry terms—colored orange—cancel each other.

From these two substitutions, it can be observed that the substitution process is initiated by a polynomial p_r of the size $|p_r| = 3(n+1)+2$; the outcome of the first substitution r_1 has a size $|r_1| = |p_r| + 3$ since one term $(-co_n)$ has been substituted for four terms; after the second substitution to replace s_n in the remainder r_1, the carry terms (colored orange) that are revealed in the polynomial g_{2n} of s_n as well as r_1 cancel each other, reducing the size of the obtained remainder $|r_2|$ than $|r_1|$ by six terms to be $|r_2| = |p_r| - 3$, four of them are carry terms and the other two are linear terms: x_n and y_n. By restricting iterative substitutions $p_{\rm r} \xrightarrow{g_{2n+1}} r_1 \xrightarrow{g_{2n}} r_2 \xrightarrow{g_{2n-1}} r_3 \cdots r_{2n+1} \xrightarrow{g_0} r_{2n+2}$ to follow the substitution order that replaces consecutively the output variables in the pair (co_i, s_i) of every adder cell, carry terms revealed in the polynomials of every pair variables cancel each other linearly—without further substitutions in these carry terms that cause exponential blow-up in the sizes of intermediate remainders. After eliminating variables of each pair (co_i, s_i), the resulted remainder size $|r_{2n-2i+2}| = |p_r| - 3(n-i+1)$ is always less than the maximum size $|p_r| + 3 = 3(n+1) + 5$ which is the size of the first remainder r_1. Thus it has been justified that the verification complexity of a SCN of a 3-bit adder array is bounded in space by $\mathcal{O}(n)$ under the substitution order that obliges consecutive substations of the variables in each pair (co_i, s_i). ■

Lemma 7. *For the multiplication function F implemented as a tuple of Boolean functions $F = (z_0, \cdots, z_{2n-1})$ with a set of primary inputs $\{x_0, \cdots, x_{n-1}, y_0, \cdots, y_{n-1}\}$ and modeled by a SCN, let the SCN model of F consists of arrays $S_i = (s_{i,0}, s_{i,1}, \cdots, s_{i,ln_i})$ which are built from adder cells $Fa_{i,w} = (s_{i,w}, co_{i,w})$, the complexity to verify the SCN model of F wrt. its polynomial specification $p_{\rm r} := -\sum_{k=0}^{2n-1} 2^k z_k + \sum_{k=0}^{n-1} 2^k x_k \cdot \sum_{k=0}^{n-1} 2^k y_k$ is bounded in space by $\mathcal{O}(n^2)$, as long as the variables in the list $\{co_{i,0}, s_{i,0}, co_{i,1}, s_{i,1}, \cdots, co_{i,ln_i}, s_{i,ln_i}\}$ of each array S_i are substituted consecutively according to the order $co_{i,ln_i} > s_{i,ln_i} > co_{i,ln_i-1} > s_{i,ln_i-1} > \cdots > co_{i,0} > s_{i,0}$.*

Proof. Multiplication function can be implemented as successive arrays of adders to sum its partial products and generate the outputs of the multiplier— named array multiplier, which can be modeled according to the following polynomials:

$$p_0 := -\sum_{k=0}^{2n-1} 2^k z_k + (2y_0x_1 + \sum_{k=2}^{2n-2} 2^k s_{0,k}) + (\sum_{k=0}^{n-1} 2^k y_k x_0 + \sum_{k=1}^{n-1} 2^{k+n-1} y_{n-1}x_k),$$

$$p_j := -\sum_{k=j+1}^{2n-j-1} 2^k s_{j,k} + (2^{j+1}y_0x_{j+1} + \sum_{k=j+2}^{2n-j-2} 2^k s_{j+1,k}) + (\sum_{k=1}^{n-j-1} 2^{k+j} y_k x_j + \sum_{k=j+1}^{n-1} 2^{k-j+n-1} y_{n-j-1}x_k),$$

$$p_{n-2} := -\sum_{k=n-1}^{n+1} 2^k s_{n-2,k} + (2^{n-1}y_0x_{n-1}) + (2^{n-1}y_1x_{n-2} + 2^n y_1x_{n-1}),$$

for $1 \leq j \leq n-3$. Every polynomial p_i is a canonical representation of an array of 3-bit adders $S_i = (s_{i,0}, s_{i,1}, \cdots, s_{i,ln_i})$ implemented by a set of polynomials $G_i = \{g_{i,0}, g_{i,1}, \cdots, g_{i,2ln_i+1}\}$ as shown in Lemma 6, where $ln_i + 1$ is the length of the array S_i. The arrays are ordered from outputs to inputs of the multiplier (reverse topological order). The first array in the order as well as the longest array is S_0 which is represented canonically by the polynomial p_0 with $2n$ outputs ($ln_0 = 2n-1$) and two operands sets: $\{0, y_0x_1, s_{0,2}, s_{0,3}, \cdots, s_{0,2n-2}\}$ together with $\{y_0x_0, y_1x_0, \cdots, y_{n-1}x_0, y_{n-1}x_1, \cdots, y_{n-1}x_{n-1}\}$. The last and the shortest 3-bit adder array S_{n-2} is described by polynomial p_{n-2}, S_{n-2} has three outputs and gets two operands sets: $\{0, y_0x_{n-1}\}$ and $\{y_1x_{n-2}, y_1x_{n-1}\}$. In between arrays S_j have the operands sets: $\{y_0x_{j+1}, s_{j+1,j+2}, \cdots, s_{j+1,2n-j-2}\}$ together with $\{y_1x_j, \cdots, y_{n-j-1}x_j, y_{n-j-1}x_{j+1}, \cdots, y_{n-j-1}x_{n-1}\}$. The multiplier is constructed by feeding the inputs of each array S_{j-1} the outputs of the array S_j.

To verify a SCN of a multiplier, the IMT procedure divides sets G_i wrt. the multiplier specification polynomial p_r. By dividing each set G_i, variables of the array S_i—the outputs of its adder cells which are in the list $\{co_{i,0}, s_{i,0}, co_{i,1}, s_{i,1}, \cdots, co_{i,ln_i}, s_{i,ln_i}\}$—will be substituted. Since every polynomial p_i is a canonical representation for the set G_i of an array S_i, iterative divisions wrt. G_i can be thought of as one division wrt. p_i. Therefore, we express iterative divisions $\xrightarrow{G_i}_+ r_{i+1}$ restricted to substitute consecutively variables of S_i by one division $\xrightarrow{p_i} r_{i+1}$.

The first division of p_r wrt. G_0 of the array S_0 is formulated as

$$p_r \xrightarrow{p_0} r_1 := \sum_{k=0}^{n-1} 2^k x_k \cdot \sum_{k=0}^{n-1} 2^k y_k - (2y_0x_1 + \sum_{k=2}^{2n-2} 2^k s_{k,0}) - (\sum_{k=0}^{n-1} 2^k y_k x_0 + \sum_{k=1}^{n-1} 2^{k+n-1} y_{n-1} x_k) = \sum_{k=1}^{n-1} 2^k x_k \cdot \sum_{k=0}^{n-2} 2^k y_k - 2y_0x_1 - \sum_{k=2}^{2n-2} 2^k s_{k,0},$$

which can be thought of as subtracting p_0 from p_r. The next group of substitution iterations divides the set G_1 of the array S_1, subtracting the polynomial p_1 from r_1, which is described as

$$r_1 \xrightarrow{p_1} r_2 := \sum_{k=1}^{n-1} 2^k x_k \cdot \sum_{k=0}^{n-2} 2^k y_k - 2y_0x_1 - 2^2 y_0x_2 - \sum_{k=3}^{2n-4} 2^k s_{k,2} - \sum_{k=1}^{n-2} 2^{k+1} y_k x_1 - \sum_{k=2}^{n-1} 2^{k+n-2} y_{n-2} x_k = \sum_{k=2}^{n-1} 2^k x_k \cdot \sum_{k=0}^{n-3} 2^k y_k - 2^2 y_0x_2 - \sum_{k=3}^{2n-4} 2^k s_{k,2}.$$

These groups of iterative substitutions can be expressed by

$$p_r \xrightarrow{p_0} r_1 \xrightarrow{p_1} r_2 \cdots \xrightarrow{p_{n-3}} r_{n-2} := \sum_{k=n-2}^{n-1} 2^k x_k \cdot \sum_{k=0}^{1} 2^k y_k - 2^{n-2} y_0 x_{n-2} - \sum_{k=m-1}^{n+1} 2^k s_{k,n-2} \xrightarrow{p_{n-1}} r_{n-1} := \sum_{k=n-2}^{n-1} 2^k x_k \cdot \sum_{k=0}^{1} 2^k y_k - 2^{n-2} y_0 x_{n-2} - 2^{n-1} y_0 x_{n-1} - 2^{n-1} y_1 x_{n-2} - 2^n y_1 x_{n-1} = 0,$$

terminating with a zero remainder.

As proved in Lemma 6, the complexity of dividing every set G_i of an array of adder cells is bounded in space by $\mathcal{O}(ln_i + 1)$, as long as variables of every pair $(co_{w,i}, s_{w,i})$ are substituted consecutively. This condition holds if the variables of each S_i are substituted consecutively according to the order $co_{i,ln_i} > s_{i,ln_i} > co_{i,ln_i-1} > s_{i,ln_i-1} > \cdots > co_{i,0} > s_{i,0}$. Under this substitution order the computation complexity of dividing any set G_i is bounded by $\mathcal{O}(2n)$ since the longest array S_0 has the size $2n-1$. By subtracting polynomials p_i from p_r which has the size n^2+n according to the reverse topological order (from outputs to inputs), the size of every resulted remainder r_i is reduced than $|p_r|$ by $|r_i| = |p_r| - 2n \cdot i + i^2 - i$. Because all $|r_i| < |p_r| < n^2 + n$ and the complexity of dividing G_i is bounded by $\mathcal{O}(2n)$, the maximum size of a remainder after any substitution step is less than $n^2 + n$. Hence the complexity of verifying a multiplier modeled as SCN is bounded in space by $\mathcal{O}(n^2)$ under a specific substitution order. ■

Since the adder cells in SCNs are not limited to 3-bit adders, it is practicable to implement the integer multiplication function by other types of cell adders such as 5-bit adders which have three outputs. At such cases, the outputs in the tuple of adder cells functions $Fa = (co_0, \cdots, co_m, s)$ must be substituted consecutively to cancel carry terms revealed within polynomials of these adder cells.

Example 11. *A 5-bit adder cell (e.g., a 4:2 compressor)* $Fa = (co_0, co_1, s)$ *has five inputs* $\{x_1, x_2, x_3, x_4, x_5\}$ *and three outputs, once it is described by three polynomials—one for every output—the carry terms are revealed as follows:*
$g_2 : -co_1 \boxed{-2x_3x_2x_1 + x_3x_2 + x_3x_1 + x_2x_1}$
$g_0 : -co_0 \boxed{-8x_5x_4x_3x_2x_1 + 4x_5x_4x_3x_2 + 4x_5x_4x_3x_1 + 4x_5x_4x_2x_1 + 4x_5x_3x_2x_1}$
$\boxed{+4x_4x_3x_2x_1 - 2x_5x_4x_3 - 2x_5x_4x_2 - 2x_5x_4x_1 - 2x_5x_3x_2 - 2x_5x_3x_1 - 2x_5x_2x_1}$
$\boxed{-2x_4x_3x_2 - 2x_4x_3x_1 - 2x_4x_2x_1 + x_5x_4 + x_5x_3 + x_5x_2 + x_5x_1 + x_4x_3 + x_4x_2}$
$\boxed{+x_4x_1}$
$g_0 : -s \boxed{+16x_5x_4x_3x_2x_1 - 8x_5x_4x_3x_2 - 8x_5x_4x_3x_1 - 8x_5x_4x_2x_1 - 8x_5x_3x_2x_1}$
$\boxed{-8x_4x_3x_2x_1 + 4x_5x_4x_3 + 4x_5x_4x_2 + 4x_5x_4x_1 + 4x_5x_3x_2 + 4x_5x_3x_1 + 4x_5x_2x_1}$
$\boxed{+4x_4x_3x_2 + 4x_4x_3x_1 + 4x_4x_2x_1 + 4x_3x_2x_1 - 2x_5x_4 - 2x_5x_3 - 2x_5x_2 - 2x_5x_1}$
$\boxed{-2x_4x_3 - 2x_4x_2 - 2x_4x_1 - 2x_3x_2 - 2x_3x_1 - 2x_2x_1} + x_5 + x_4 + x_3 + x_2 + x_1$.
Carry terms are those that colored blue. By substituting variables (co_0, co_1, s) *in the specification polynomial* $p_r := -2co_0 - 2co_1 - s + x_1 + x_2 + x_3 + x_4$ *during the verification process of the 5-bit adder, these carry terms cancel each other linearly, resulting in a zero remainder.*

Since Lemma 7 assumes that the multiplier is constructed from only 3-bit adder cells, we extend Lemma 7 for multipliers that contain different adder cells with arbitrary numbers of inputs.

Lemma 8. *For the multiplication function* F *implemented as a tuple of Boolean functions* $F = (z_0, \cdots, z_{2n-1})$ *with a set of primary inputs* $\{x_0, \cdots, x_{n-1}, y_0, \cdots, y_{n-1}\}$ *and modeled by a SCN, let the SCN model of* F *consists of arrays* $S_i = (s_{i,0}, s_{i,1}, \cdots, s_{i,ln_i})$ *which are built from adder cells* $Fa_{i,w} = (co_{i,w,0}, co_{i,w,1}, \cdots, co_{i,w,m}, s_{i,w})$ *with arbitrary sizes, the complexity to verify the SCN model of* F *wrt. its polynomial specification* $p_r := -\sum_{k=0}^{2n-1} 2^k z_k + \sum_{k=0}^{n-1} 2^k x_k \cdot \sum_{k=0}^{n-1} 2^k y_k$ *is bounded in space by* $\mathcal{O}(n^2)$*, as long as the variables in the list* $\{co_{i,0,0}, \cdots, co_{i,0,m}, s_{i,0}, co_{i,1,0}, \cdots, co_{i,1,m}, s_{i,1}, \cdots, co_{i,ln_i,0}, \cdots, co_{i,ln_i,m}, s_{i,ln_i}\}$ *of each array* S_i *are substituted consecutively according to the order* $co_{i,ln_i,m} >$

$\cdots co_{i,ln_i,0} > s_{i,ln_i} > co_{i,ln_i-1,m} > \cdots > co_{i,ln_i-1,0} > s_{i,ln_i-1} > \cdots > co_{i,0,m} > \cdots > co_{i,0,0} > s_{i,0}$.

Proof. The substitution order $co_{i,ln_i,m} > \cdots co_{i,ln_i,0} > s_{i,ln_i} > co_{i,ln_i-1,m} > \cdots > co_{i,ln_i-1,0} > s_{i,ln_i-1} > \cdots > co_{i,0,m} > \cdots > co_{i,0,0} > s_{i,0}$ obligates the verification process to substitute consecutively outputs of each adder cell $Fa_{i,w} = (co_{i,w,0}, \cdots, co_{i,w,m}, s_{i,w})$ in F, which bounds the verification complexity of each adder array S_i by $\mathcal{O}(n)$ since carry terms that cause the exponential complexity are canceled under this obligated substitution order. Hence, as in Lemma 7, the complexity of verifying an integer multiplier modeled as a SCN is bounded in space by $\mathcal{O}(n^2)$ regardless of the size of the compounded adder cells. ■

In the summary, rewriting the bit-level models of integer functions into SCNs is crucial to circumvent the exponential complexity of these functions, otherwise, the carry terms will only be eliminated after reducing them to the input variables, which leads to an exponential increase in the number and size of the nonlinear terms. The rewriting into SCNs is effective for the verification of basic multiplier architectures, i.e., multipliers with simple partial products generators and ripple carry adders in the last addition stage. However the IMT procedure fails to verify more complex architectures even if they modeled as SCNs because their models contain, besides carry terms, redundant nonlinear terms named vanishing monomials. In the next section, we will provide an explanation of this limitation, which forms the basis of our proposed model rewriting algorithm that lifts bit-level multipliers into SCNs together with removing vanishing monomials from models of SCNs.

4.3 PROBLEM OF VANISHING MONOMIALS

Simplifying the circuit model by rewriting them into SCNs is not sufficient for the verification of integer multipliers that consist of parallel adders or Booth recoding. The main reason are vanishing monomials—monomials that always evaluate to zero—which appear in every algebraic model of these complex multipliers. Unfortunately, the IMT cannot cancel these vanishing monomials before substituting them with primary input variables. Some of the vanishing monomials have the property that representing them by input variables will increase the number of intermediate monomials exponentially, making the computation unfeasible. In this and the following section, we illustrate the vanishing monomials limitation with two examples: a PPA adder and a Booth partial product cell;

and show how to overcome this problem by a new rewriting scheme enhanced by logic reduction.

Example 12. *Consider a circuit model of a 3-bit PPA:*[1]

$$s_3 = c_2 \quad \Longrightarrow \quad g_1 := -s_3 + c_2$$
$$c_2 = d_2 \vee (e_2 \wedge d_1) \vee (e_2 \wedge e_1 \wedge d_0) \quad \Longrightarrow \quad g_2 := -c_2 + e_2d_2e_1d_1d_0 - e_2e_1d_1d_0 - e_2d_2e_1d_0 - e_2d_2d_1 + e_2e_1d_0 + e_2d_1 + d_2$$

$$\begin{aligned}
s_2 = e_2 \oplus c_1 &\Longrightarrow g_3 := -s_2 - 2c_1e_2 + c1 + e_2\\
c_1 = d_1 \vee (e_1 \wedge d_0) &\Longrightarrow g_4 := -c_1 - e_1d_1d_0 + e_1d_0 + d_1\\
s_1 = e_1 \oplus c_0 &\Longrightarrow g_5 := -s_1 - 2c_0e_1 + c_0 + e_1\\
c_0 = d_0 &\Longrightarrow g_6 := -c_0 + d_0\\
s_0 = e_0 &\Longrightarrow g_7 := -s_0 + e_0\\
e_2 = x_2 \oplus y_2 &\Longrightarrow g_8 := -e_2 - 2y_2x_2 + y_2 + x_2\\
d_2 = x_2 \wedge y_2 &\Longrightarrow g_9 := -d_2 + y_2x_2\\
e_1 = x_1 \oplus y_1 &\Longrightarrow g_{10} := -e_1 - 2y_1x_1 + y_1 + x_1\\
d_1 = x_1 \wedge y_1 &\Longrightarrow g_{11} := -d_1 + y_1x_1\\
e_0 = x_0 \oplus y_0 &\Longrightarrow g_{12} := -e_0 - 2y_0x_0 + y_0 + x_0\\
d_0 = x_0 \wedge y_0 &\Longrightarrow g_{13} := -d_0 + y_0x_0.
\end{aligned}$$

s_i is the sum bit, c_i is the carry bit, and for every input bits x_i, y_i, there is a generation bit d_i and a propagation bit e_i. The vanishing monomials in this model are colored red. As an example consider the vanishing monomial $e_1d_1d_0$ of polynomial g_4. Substituting e_1 and d_1 in this monomial yields $e_1d_1d_0 \xrightarrow{g_{10}} -2d_1d_0y_1x_1 + d_1d_0y_1 + d_1d_0x_1 \xrightarrow{g_{11}} -2d_0y_1x_1 + d_0y_1x_1 + d_0y_1x_1 = 0$. It is clear that the IMT can easily cancel this monomial. However, the corresponding monomial in the representation of the highest carry for an n-bit adder is $e_{n-1} \dots e_2e_1d_1d_0$. This follows from modeling the carry bit $c_{n-1} = d_{n-1} \vee (e_{n-1} \wedge d_{n-2}) \vee (e_{n-1} \wedge e_{n-2} \wedge d_{n-3}) \vee \dots \vee (e_{n-1} \wedge e_{n-2} \wedge \dots \wedge e_2 \wedge d_1) \vee (e_{n-1} \wedge e_{n-2} \wedge \dots \wedge e_2 \wedge e_1 \wedge d_0)$.

By substituting in this monomial according to the order $e_{n-1} > d_{n-1} > \dots > e_0 > d_0$, the number of vanishing monomials will increase from 1 to 3^{n-1} monomials with a maximum size of $2n$ variables. Consider another vanishing monomial $e_2d_2e_1d_0$ of polynomial g_2, the corresponding vanishing monomial for the carry bit c_{n-1} is $e_{n-1}d_{n-1}e_{n-2} \dots e_2e_1d_0$. By substituting in this monomial with a different order $e_0 > d_0 > \dots > e_{n-1} > d_{n-1}$ compared to the

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

previous one, the number of intermediate vanishing polynomials will increase to be about 3^{n-1} *with a maximum size of* $2n$ *variables. From these two examples, we conclude that it is hard to find a substitution order to cancel all vanishing monomials before they blow up.*

The experimental results of [99] confirm the problem of the IMT with parallel adders. Their results show that the technique cannot verify Kogge-Stone adders with more than 6 bits. Concluding our observations above, the core problem that we need to solve is the occurrence of a large number of vanishing monomials that lead to an exponential blow-up when reduced to primary input variables.

4.4 LOGIC REDUCTION WITHIN MODEL REWRITING

This section presents the integration of a logic reduction rule within model rewriting that consists of two rewriting schemes. The proposed solution eliminates vanishing monomials before they cause a blow-up and rewrites bit-level models into SCNs.

4.4.1 *Logic Reduction*

To overcome the limitation caused by vanishing monomials during the IMT, we propose to apply logic reduction during the rewriting of the Gröbner basis model in order to remove vanishing monomials before their blow-up. Looking again at Example 12, it is easy to see that the monomial $e_1 d_1 d_0$ can be removed when considering that the variable e_1 is the XOR of x_1, y_1, and the variable d_1 is the AND of x_1, y_1. Based on this structural knowledge of the circuit model, we can conclude that the monomial always evaluates to zero since $(x \oplus y) \cdot (x \wedge y) = 0$ for all x and y and therefore can be used to remove terms from polynomials. If, e.g, $f_1 = x \oplus y$ and $f_2 = x \wedge y$, any term incorporating both f_1 and f_2 can be removed. We refer to this as *XOR-AND vanishing rule*. By keeping track of the original gate function and the input variables associated to each variable, we can effectively search for monomials that satisfy the XOR-AND vanishing rule. Applying this rule will remove all the vanishing monomials of the parallel adder model shown in Example 12, and will avoid the high computation cost of the IMT.

We have published this idea in [87]. However, the XOR-AND vanishing rule requires structural knowledge of the circuit model to be applied, so that we propose to generalize this rule, in the sense that no structural knowledge is needed

and the rule is not restricted to the product of AND and XOR functions—the rule can reduce the product of any arbitrary functions.

Let X be a set of variables and let f_1 and f_2 be two Boolean functions over the variables $X_1 \subseteq X$ and $X_2 \subseteq X$, respectively, with $X_1 \cap X_2 \neq \emptyset$. If there exists exactly one assignment to f_2 such that it evaluates to true, it may be possible that f_1 is simplified to a constant value when assigning the common variables according to that assignment. To illustrate the concept consider a multiplexer function $f_1(x_1, x_2, x_3) = x_1x_3 - x_2x_3 + x_3$ and $f_2(x_1, x_2) = x_1x_2$. Clearly $f_2 = 1$, only if $x_1 = x_2 = 1$, and $f_1(1, 1, x_3) = 1$. Therefore, we conclude that $f_1f_2 = f_2$ and we can simplify polynomials accordingly. For a polynomial $g := -x_4x_5f_1f_2x_6 + x_4x_5f_2x_6$, by applying this rule on the monomial $x_4x_5f_1f_2x_6$, it is simplified to $x_4x_5f_2x_6$ and the polynomial g will be evaluated to $g := -x_4x_5f_2x_6 + x_4x_5f_2x_6 = 0$. This reduction rule is called *one assignment rule*. An approach to apply one assignment rule is as follows:

1. Searching for monomials in the algebraic model that have two variables of functions f_1 and f_2 which shared some of their inputs, such that the function f_2 has one satisfiable assignment.

2. Reducing f_1 after assigning values to shared inputs which evaluate f_2 to one.

3. If f_1 is equal to zero or one, then rewriting the monomial by substituting f_1 with its value.

The one assignment rule supersedes the XOR-AND vanishing rule since the one assignment rule is not restricted to specific Boolean functions and it does not require a structure knowledge. However, because of the wide applicability of the one assignment rule, unnecessary search for monomials that hold the rule is performed. To solve this problem, we limit the applicability of the rule to those functions f that have number of inputs less than or equal to eight. This number is chosen based on the observation, that the rule is applied within a rewriting scheme (as shown in the next subsection) which rewrites the bit-level multipliers into adder cells where the maximum size of an adder cell is eight inputs. This restriction might be refined when the one assignment rule is utilized for other applications.

4.4.2 *Rewriting Schemes*

Although the correspondence between gates in the circuits and variables in the polynomials is given, rewriting the model is crucial to reveal vanishing

monomials together with carry terms by lifting the bit-level description of the circuit into a SCN. Without rewriting the model:

1. if *no* substitution is applied, one may not see monomials that contain variables satisfying the one assignment rule as well as carry terms, and

2. if *arbitrary* substitution is applied, a blow-up may occur within the rewriting process because of substituting some variables of these nonlinear terms.

Both cases prohibit the application of the one assignment rule as well as canceling carry terms for each other during the IMT procedure. Consequently, we integrate two rewriting schemes which permit to reveal within models of SCNs vanishing monomials in addition to carry terms.

XOR rewriting

We propose the *XOR rewriting* scheme that preserves input/output variables of XOR gates, it is carried out after Gröbner bases modeling for the circuit and before applying the IMT procedure, it performs the following steps:

1. Store all variables in a list V that refer to either input and output variables of an XOR gate or to primary inputs and primary outputs. Variables that are outputs of XOR gates and given as inputs for other XOR gates will also be excluded from the list.

2. Rewrite the model by substituting some variables of the model using the rewrite rule (see Definition 17) such that the model depends only on variables in V. After each substitution, the approach to apply the one assignment rule is performed.

As shown in Figure 20, the result of this XOR rewriting are set of polynomials that model chains of XOR gates and others that model Boolean functions (combination of arbitrary logic gates with multiple inputs and one output) given as inputs to the XOR chains. In the figure, the cloud symbols refer to the combination of arbitrary logic gates, x are primary inputs, and xo are primary outputs.

Example 13. *Continue with Example 12, the algebraic model of the parallel adder after XOR rewriting is as follows:*

$s_3 = c_2 \qquad \Longrightarrow \; g_1 := -s_3 + c_2$

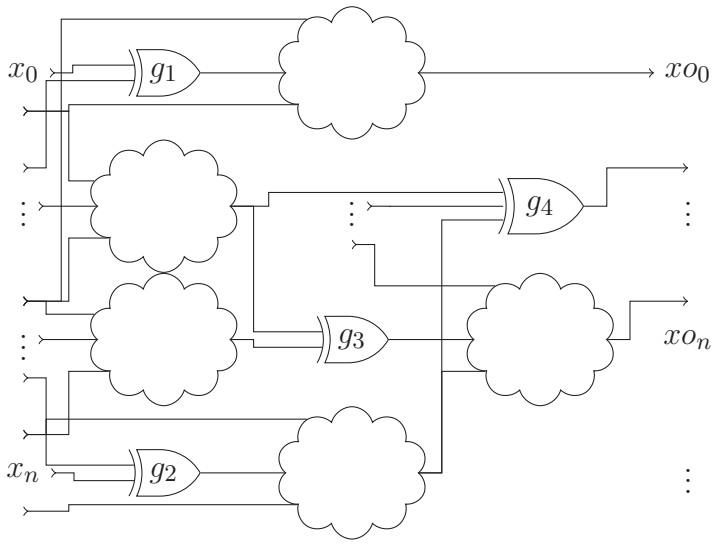

Figure 20: Schematic of Circuit Model after XOR Rewriting

$$c_2 = d_2 \vee (e_2 \wedge d_1) \vee (e_2 \wedge e_1 \wedge d_0) \implies g_2 := -c_2 + e_2e_1x_2y_2x_1y_1x_0y_0 - e_2e_1x_1y_1x_0y_0 - e_2e_1x_2y_2x_0y_0 - e_2x_2y_2x_1y_1 + e_2e_1x_0y_0 + e_2x_1y_1 + x_2y_2$$

$$s_2 = e_2 \oplus c_1 \implies g_3 := -s_2 - 2c_1e_2 + c1 + e_2$$

$$c_1 = d_1 \vee (e_1 \wedge d_0) \implies g_4 := -c_1 - e_1x_1y_1x_0y_0 + e_1x_0y_0 + x_1y_1$$

$$s_1 = e_1 \oplus c_0 \implies g_5 := -s_1 - 2c_0e_1 + c_0 + e_1$$

$$c_0 = d_0 \implies g_6 := -c_0 + x_0y_0$$

$$s_0 = e_0 \implies g_7 := -s_0 + e_0$$

$$e_2 = x_2 \oplus y_2 \implies g_8 := -e_2 - 2y_2x_2 + y_2 + x_2$$

$$e_1 = x_1 \oplus y_1 \implies g_{10} := -e_1 - 2y_1x_1 + y_1 + x_1$$

$$e_0 = x_0 \oplus y_0 \implies g_{12} := -e_0 - 2y_0x_0 + y_0 + x_0$$

XOR rewriting have eliminated all variables that are not inputs or outputs of chain of XOR gates, therefore, it keeps only variables in the list $\{e_0, e_1, e_2, c_0, c_1, c_2\}$. *This leads to lifting the bit-level description into polynomials expressing mainly the functions of the sum and carry variables of adder cells existing within the circuit, which exposes vanishing monomials same as in the previous model of Example 12.*

Also, we have observed that the XOR rewriting is efficient to reveal vanishing monomials of the Booth partial product cell, this is demonstrated by the following example.

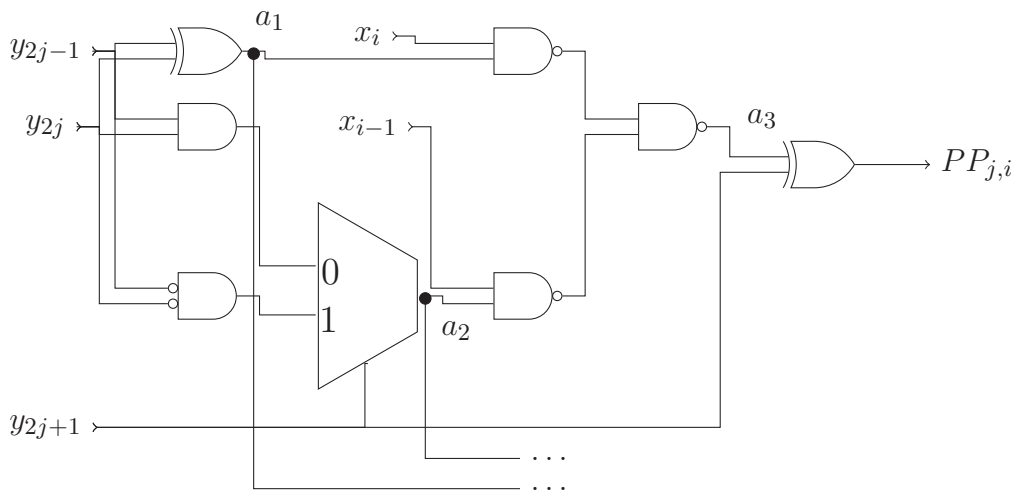

Figure 21: Booth Partial Product Cell

Example 14. *Figure 21 shows a Booth partial product cell, which is a building block of many efficient multiplier circuits. The cell has mainly two parts. The first part is Booth's encoder with output variables* a_1 *and* a_2*. The second part is the generator which generates the partial product* $PP_{j,i}$*. Note that* a_1 *and* a_2 *are utilized to generate further partial product bits, so they are considered multiple fanout variables. Overall, the fanout variables of the circuit are* $a_1, a_2, PP_{j,i}$*, while the XOR variables are* $a_1, a_3, PP_{j,i}$ *(remember also inputs of an XOR are stored in* V*, here* x_1*). The algebraic model of the circuit after XOR rewriting will be:*

$g_1 := -PP_{j,i} - 2a_3y_{2j+1} + a_3 + y_{2j+1}$

$g_2 := -a_3 + a_1y_{2j+1}y_{2j}x_ix_{i-1} + a_1y_{2j+1}y_{2j-1}x_ix_{i-1} - a_1y_{2j+1}x_ix_{i-1} + a_1x_i - y_{2j+1}y_{2j}x_{i-1} - y_{2j+1}y_{2j-1}x_{i-1} + y_{2j+1}x_{i-1} + y_{2j}y_{2j-1}x_{i-1} - a_1y_{2j}y_{2j-1}x_ix_{i-1}$

$g_3 := -a_1 - 2y_{2j}y_{2j-1} + y_{2j} + y_{2j-1}$.

The vanishing monomial $a_1y_{2j}y_{2j-1}x_ix_{i-1}$ *will appear. It contains* a_1*, which is the XOR of* y_{2j} *and* y_{2j-1}*, and the product* $y_{2j}y_{2j-1}$*, which is the AND of* y_{2j} *and* y_{2j-1}*.*

Common rewriting

To reveal carry terms, a rewriting scheme named *fanout rewriting* [39], has been proposed based on the fanouts of the circuit gates, such that the model terms will depend only on shared variables. This dependency increases the chance of exposing terms with common monomials, which is the main feature

of carry terms. The fanout rewriting is performed in two steps: 1) It finds the gates that have multiple fanouts and stores the corresponding output variables in a list, 2) It substitutes all variables that are not in this list, such that the model will depend only on fanouts, primary inputs, and primary outputs. The rewriting allows revealing carry terms that cancel each other during the IMT without exponential complexity. This positive effect is not provided by XOR rewriting, making the verification inefficient if only XOR rewriting is applied. Hence, we propose to carry out a further rewriting called *common rewriting*, which is similar to fanout rewriting, after XOR rewriting. Common rewriting exposes carry terms by making the polynomials depend on shared variables. It rewrites the model obtained from XOR rewriting such that the polynomials depend only on variables that are used in more than one polynomial. This is very similar to fanout rewriting, but since we are no longer working on the original circuit model, one cannot strictly speak of fanout variables.

Example 15. *Continue with Example 12 and Example 13, the algebraic model of the parallel adder after removing vanishing monomials is as follows:*

$g_1 := -s_3 + c_2$
$g_2 := -c_2 + e_2e_1x_0y_0 + e_2x_1y_1 + x_2y_2$
$g_3 := -s_2 - 2c_1e_2 + c1 + e_2$
$g_4 := -c_1 + e_1x_0y_0 + x_1y_1$
$g_5 := -s_1 - 2c_0e_1 + c_0 + e_1$
$g_6 := -c_0 + x_0y_0$
$g_7 := -s_0 + e_0$
$g_8 := -e_2 - 2y_2x_2 + y_2 + x_2$
$g_{10} := -e_1 - 2y_1x_1 + y_1 + x_1$
$g_{12} := -e_0 - 2y_0x_0 + y_0 + x_0.$

Since the variables in the list $\{e_0, c_0, c_1, c_2\}$ *are not used as inputs for more than one polynomial in the model, the common rewriting eliminates these variables deriving a new model as follows:*

$g_1 := -s_3 + e_2e_1x_0y_0 + e_2x_1y_1 + x_2y_2$
$g_3 := -s_2 - 2e_2e_1x_0y_0 - 2e_2x_1y_1 + e_1x_0y_0 + x_1y_1 + e_2$
$g_5 := -s_1 - 2e_1x_0y_0 + x_0y_0 + e_1$
$g_7 := -s_0 - 2x_0y_0 + y_0 + x_0$
$g_8 := -e_2 - 2x_2y_2 + y_2 + x_2$
$g_{10} := -e_1 - 2x_1y_1 + y_1 + x_1.$

The resulted model after common rewriting reveals carry terms which are colored similarly.

4.4.3 *Overall Algorithm*

Both XOR rewriting and common rewriting follow two steps, which are identifying a set of variables and then substituting all remaining variables. Hence, the rewriting can be explained by a generalized algorithm, named *Gröbner Rewriting* (GB-Rew), illustrated in algorithm 2. It substitutes the variables that are not in V using the rewrite rule (see Definition 17). Additionally, within the GB-Rew, monomials are removed from the model using the one assignment rule after every substitution.

The rewriting is performed by substituting variables of every single polynomial in the model. The polynomials are considered in reverse order of their leading monomial variables. i.e., for a model of two polynomials $g_1 := x_1 + \text{tail}(g_1)$ and $g_2 := x_2 + \text{tail}(g_2)$ with monomial ordering (The variables are ordered according to the reverse topological order of the circuit, as explained in Subsection 2.2.3). $x_2 > x_1$, the polynomial g_1 will be considered first.

Within a single polynomial, the substitution order of the variables plays a role in enhancing the time performance of the rewriting. The substitution order is chosen according to the number of terms in the tail part of their polynomials. For example, assume a single polynomial g_s has a term x_1x_2, to rewrite g_s, x_1 and x_2 are required to be substituted by the rewriting rule. Based on their polynomials $g_1 := x_1 + \text{tail}(g_1)$ and $g_2 := x_2 + \text{tail}(g_2)$, variable x_1 is substituted before x_2 if the number of terms (n_1) in $\text{tail}(g_1)$ is smaller than the number of terms (n_2) in $\text{tail}(g_2)$. In the case that the two substitutions yield to cancel some intermediate terms (number of terms of g_s will be increased by less than $n_1 \cdot n_2$ after the two substitutions), following the proposed order reduces the maximum number of terms of the intermediate forms of the polynomial g_s.

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

The overall rewriting algorithm is carried out before calling the IMT procedure by executing successively XOR rewriting and common rewriting schemes to obtain from the bit-level description of the multiplier a SCN model. This is also illustrated by Algorithm 3, and is referred as model rewriting.

4.4.4 *Discussion*

So far, we have demonstrated that the proposed integration between XOR-rewriting scheme and common rewriting scheme brings implicitly the bit-level

Algorithm 2 Gröbner Basis Rewriting (GB-Rew)

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

Algorithm 3 Model Rewriting

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

description into a SCN by exposing carry terms as well as purifying the model from vanishing monomials, but *why* such integration is capable of doing this task. The proposed rewriting schemes exploit a common property of different multiplier architectures: they are composed of adder cells such as 3-bit adders or 5-bit adders, which generate sum and carry bits, where the sum bit is the output of a chain of XOR gates, while they differ in implementations of the function of carry bits. For a given gate netlist, the XOR rewriting detects chains of XOR gates and assumes that a sum bit of an adder component is the result of every single chain. For an adder cells with n inputs x_i, the XOR rewriting eliminates the internal variables of the cell, consequently, it describes the function of a sum bit s and the functions of output carry bits co_j by polynomials mapping them directly to input variables x_i. Substituting only these internal variables reveals

Algorithm 4 IMT with Substitution Rules

Require: Specification polynomial p_{r}, circuit polynomials $G = \{g_1, g_2, \ldots, g_s\}$
Ensure: Remainder r

1: $V \leftarrow OrderedPolynomialVariables(p_{\mathrm{r}}, G)$ { Reverse Topological Order}
2: $r \leftarrow p_{\mathrm{r}}$
3: $x \leftarrow SelectingVariable(r, V)$ { Searching for a variable satisfying one of substitution rules}
4: **while** $x \notin PrimaryInputs$ **do**
5: Choose $g_t \in G$ such that $\mathrm{lm}(g_t) = x$
6: $r \xrightarrow{g_t}_{+} r$
7: $x \leftarrow SelectingVariable(r, V)$
8: **end while**

vanishing monomials in the polynomials describing the carry functions and in the same time it keeps these monomials without further substitutions that may cause a blow-up. Applying common rewriting after XOR rewriting yields the revealing of carry terms which are shared among sum and carry polynomials. Thus the combination of these two rewriting schemes succeeds to rewrite a large class of bit-level multiplier architectures into models of SCNs.

Since the computational complexity of verifying SCNs is polynomial under a specific substitution order, as proved by Lemma 8, the proposed model rewriting algorithm is the first stage toward a full automated verification for bit-level multiplier architectures. The second stage is an efficient IMT procedure which applies the substitution order that averts an exponential blow-up in the size of the verification problem. In the next section, we introduce *substitution rules* that qualify the IMT for performing such a task.

4.5 IDEAL MEMBERSHIP TESTING

The IMT can be thought of as an algorithm that is given in every iteration a polynomial r_i and set of variables $V = \{x_0, x_1 \cdots, x_n\}$ that are related to a set of polynomials G (the Gröbner basis Model). The task of the IMT is to substitute in r_i one of the variables $x \in V$ for its corresponding polynomial $g \in G$ using the rewrite rule, ensuing a new polynomial r_{i+1} which is together with V are the inputs of the next iteration. In each iteration i, it decides which variable is better to be substituted, thereby it circumvents a potential blow-up in the sizes of resulted polynomials at next iterations. The IMT terminates when there is no further substitutions that can be performed, i.e., the resulted poly-

nomial depends only on primary inputs of the model. Under the assumption that there is a substitution order that solves the verification problem in a polynomial space, the challenge of the IMT procedure is to find this substitution order.

For the verification problem of the multiplier, in [23, 39], a fixed substitution order is given to the IMT based on the reverse topological order of the circuit, which restricts variables that have the same level and depend on common inputs to be substituted consecutively. However, this given substitution order does not solve the problem effectively in case of complex multipliers—it is not always the optimal order. Based on our analysis to the verification of SCNs of multipliers in Section 4.2, we support this fixed substitution order by substitution rules that qualify the IMT procedure to take substitution decisions that are consistent with Lemma 8. To guarantee a tractable verification process for SCNs of multipliers, Lemma 8 stipulates to substitute consecutively: 1) output variables of each adder cell in SCNs and 2) outputs of each subset of adder cells that build an array, i.e., for two successive adder cells in an array of SCNs with tuples $F_a = (co_0, \cdots, co_m, s)$ and $F_{\hat{a}}(\hat{co}_0, \cdots, \hat{co}_m, \hat{s})$, the optimal substitution order is $co_0 > \cdots > co_m > s > \hat{co}_0, \cdots, \hat{co}_m > \hat{s}$. Finding this optimal substitution order will be feasible if boundaries of arrays and their constitutive adder cells are known. Unfortunately, this is not the case with SCNs generated by the model rewriting algorithm, whereas boundaries of their components are not identified. Furthermore, we have observed that for some complex multiplier architectures, the model rewriting algorithm is not capable of revealing all carry terms within adder cells of SCNs. This means that the given input to the IMT is not always a pure SCN, this implies that there are some nonlinear terms will not be canceled without eliminating at least one variable from such terms. To overcome these problems, we propose substitution rules that aim to deduce the optimal substitution order without a knowledge about boundaries of a given non-pure SCN, whereas the fixed substitution order of [23] is not efficient to deal with complex multipliers even after rewriting their models into SCNs. This fixed order is applicable only for multipliers consisting mainly of full adders arrays, where the outputs of each full adder have the same level and depend on common inputs, thus the fixed order substitutes outputs of each full adder consecutively, as it is stipulated by Lemma 8.

Our proposed substitution rules that qualify the IMT to find the proper substitution order for non-pure SCNs are as follows:

1. Rule #1 obliges the IMT to substitute first variables in linear terms (terms with a single variable) once they are exposed in a polynomial r_i before

those in nonlinear terms. In case that a variable is included in nonlinear terms as well as a linear term, only the linear term will be eliminated.

2. Rule #2 is applied when there are no variables satisfying rule #1. It prioritizes substituting a variable x_s over other variables of nonlinear terms in r_i. If there is a nonlinear term t_1 which is the product of variables in the set X_1, where $x_s \in X_1$, and there is another nonlinear term t_2 with set of variables X_2, such that $X_1 = X_2 \cup \{x_s\}$ (i.e., the difference between the two sets is only x_s).

3. Rule #3 is valid when Rule #1 and Rule #2 are not possible. It gives a higher priority for a variable $x_s \in X_1$ of a nonlinear term t_1, if there is another nonlinear term t_2 with set of variables X_2, such that $X_1 - \{x_s\} \subset X_2$ (i.e., in addition to x_s, X_2 has more variables that are not in X_1).

4. If the IMT has to choose between more than one variable that all satisfy one of the previous rules, Rule #4 selects the highest variable in the reverse topological order. Rule #4 is also applied when there are no more variables satisfying any of the previous rules.

To discuss the efficiency of these rules, note that the outputs of adder cells are expressed by linear terms, e.g., the tuple of adder cells outputs $(s, co_0, \cdots, co_n)$ is represented in a polynomial r_i as $-2co_n - \cdots - 2co_0 - s$. Rule #1 exploits this feature to identify output variables of adder cells and substitute them first. The combination between rule #1 and rule #4 prohibits to eliminate variables from an array of adder cells $\hat{S}$ before finishing the elimination of all variables in the previous array S since the reverse topological order classifies the variables of $\hat{S}$ and S into two separated groups. Thus such combination qualifies the IMT to take substitution decisions that are consistent with Lemma 8.

Rule #2 as well as Rule #3 deal with nonlinear terms that are exposed since the given model to the IMT is not a pure SCN. The rules increase the chances of canceling the terms to each other by prohibiting the elimination of shared variables among these nonlinear terms, whereas terms t_1 and t_2 cancel each other when they are the products of the same set of variables X and their coefficients differ only in signs.

The rules are applied by modifying the original IMT algorithm (see Algorithm 1). As shown in Algorithm 4, the variables in the list V of the model G are sorted based on the reverse topological order. Then IMT is given in each iteration a remainder r and the ordered list V to select one variable $x \in V$ that

satisfies one of substitution rules. The algorithm terminates when all variables of r are primary inputs.

In the summary, we have supported the IMT procedure by substitution rules that are independent of the structural knowledge of the circuit, the rules enhance the applicability as well as the time performance of the IMT. In the next section, the chapter discusses the second input of the IMT which is the specification polynomial.

4.6 SPECIFICATION POLYNOMIAL

In the symbolic computation technique over a Boolean ring, the verification process of a multiplier is performed by dividing the algebraic module of the multiplier circuit wrt. the polynomial $p_{\mathrm{r}} := \sum_{i=0}^{2n-1} -2^i s_i + \sum_{i=0}^{n-1} 2^i x_i \cdot \sum_{i=0}^{n-1} 2^i y_i$. However, we have observed a problem with such a formulation for the p_{r}, it does not match mathematically with all models of multipliers. The problem is remarked with specific types of multipliers such as those consisting of Booth partial products or redundant binary addition trees. The problem is also seen during the verification of partial multipliers where not all $2n$ outputs are verified, but rather $m < 2n$ outputs are involved in the verification process. The problem is manifested by nonlinear terms that the absolute values of their coefficients are larger than 2^m, where m is the number of the multiplier outputs which are under the verification. This problem can be illustrated by the following example.

Example 16. *Consider the model of a part of a multiplier (3-output bits) verified against the specification polynomial* $p_{\mathrm{r}} := -4z_2 - 2z_1 - z_0 + 4y_2x_0 + 4y_1x_1 + 4x_2y_0 + 2y_1x_0 + 2x_1y_0 + y_0x_0$. *The model of this multiplier part is as follows:*
$z_2 = c_1 \oplus (y_2 \wedge x_0) \oplus (y_1 \wedge x_1) \oplus (y_0 \wedge x_2) \implies g_4 := -z_2 - 8c_1y_2x_2y_1x_1y_0x_0 + 4c_1y_2y_1x_1x_0 + 4c_1x_2x_1y_1y_0 + 4c_1y_2x_2y_0x_0 + 4y_2x_2y_1x_1y_0x_0 - 2c_1y_2x_0 - 2c_1y_1x_1 - 2c_1x_2y_0 - 2y_2y_1x_1x_0 - 2y_2x_2y_0x_0 - 2x_2y_1x_1y_0 + c_1 + y_2x_0 + y_1x_1 + y_0x_2$
$c_1 = (y_1 \wedge x_0) \wedge (y_0 \wedge x_1) \implies g_3 := -c_1 + y_1x_1y_0x_0$
$z_1 = (y_1 \wedge x_0) \oplus (y_0 \wedge x_1) \implies g_2 := -z_1 - 2y_1x_1y_0x_0 + y_1x_0 + y_0x_1$
$z_0 = (y_0 \wedge x_0) \implies g_1 := -z_0 + y_0x_0.$
The verification process is performed by iterative divisions:
$p_{\mathrm{r}} \xrightarrow{g_4} r_1 := 32c_1y_2x_2y_1x_1y_0x_0 - 16c_1y_2y_1x_1x_0 - 16c_1x_2x_1y_1y_0 - 16c_1y_2x_2y_0x_0 - 16y_2x_2y_1x_1y_0x_0 + 8c_1y_2x_0 + 8c_1y_1x_1 + 8c_1x_2y_0 + 8y_2y_1x_1x_0 + 8y_2x_2y_0x_0 + 8x_2y_1x_1y_0 - 4c_1 - 2z_1 - z_0 + 2y_1x_0 + 2x_1y_0 + y_0x_0$
$\xrightarrow{g_3} r_2 \xrightarrow{g_2} r_3 \xrightarrow{g_1} r_4 : -16y_2y_1x_1y_0x_0 - 16x_2y_1x_1y_0x_0 + 8y_2y_1x_1y_0x_0 + 8y_1x_1y_0x_0 + 8x_2y_1x_1y_0x_0 + 8y_2y_1x_1x_0 + 8y_2x_2y_0x_0 + 8x_2y_1x_1y_0.$

The final remainder r_4 is not equal to zero, it consists of nonlinear terms that have coefficients with absolute value larger than 2^2. Therefore, the specification does not match the model of the partial multiplier.

In case of redundant multipliers such as those that include Booth recoding or redundant trees, the accumulator trees of these n-bit multipliers are not only fed partial products but also other input bits [62], so that these trees should produce mathematically more than $2n$ outputs, however, only the least $2n$ outputs are considered as the multiplier outputs. This implies that parts of the addition trees of redundant multipliers are involved in the verification process, while other outputs are discarded. Thus verifying $2n$ outputs of n-bit redundant multipliers has the same problem like verifying a partial multiplier.

To overcome this problem, we propose the idea of adding modulo 2^{m+1} to the specification of integer multipliers, such that the specification matches partial multipliers and redundant multipliers. Modulo 2^{m+1} is performed by removing from the final remainder r the terms that their coefficients are multiples of 2^m. Because of that the specification polynomial is formulated as:

$$\sum_{i=0}^{m} -2^i s_i + \sum_{i=0}^{n-1} 2^i x_i \cdot \sum_{i=0}^{n-1} 2^i y_i \mod 2^{m+1}.$$

Example 17. *Continue with Example 16, but modify the specification polynomial to be* $p_{\mathrm{r}} := -4z_2 - 2z_1 - z_0 + 4y_2x_0 + 4y_1x_1 + 4x_2y_0 + 2y_1x_0 + 2x_1y_0 + y_0x_0 \mod 8$. *The remainders of iterative divisions will be as follows:*

$p_{\mathrm{r}} \xrightarrow{g_4} r_1 := 32c_1y_2x_2y_1x_1y_0x_0 - 16c_1y_2y_1x_1x_0 - 16c_1x_2x_1y_1y_0$
$-16c_1y_2x_2y_0x_0 - 16y_2x_2y_1x_1y_0x_0 + 8c_1y_2x_0 + 8c_1y_1x_1 + 8c_1x_2y_0 + 8y_2y_1x_1x_0$
$+8y_2x_2y_0x_0 + 8x_2y_1x_1y_0 - 4c_1 - 2z_1 - z_0 + 2y_1x_0 + 2x_1y_0 + y_0x_0 \mod 8 =$
$-4c_1 - 2z_1 - z_0 + 2y_1x_0 + 2x_1y_0 + y_0x_0 \mod 8$
$\xrightarrow{g_3} r_2 \xrightarrow{g_2} r_3 \xrightarrow{g_1} r_4 := 0$. *The divisions terminate with zero remainder, which means that the modified specification matches the partial multiplier.*

4.7 EXPERIMENTAL RESULTS

The enhanced symbolic computation technique by the proposed algorithms in this chapter is named (SC-LR) since it integrates logic reduction with the symbolic computation. SC-LR and the algorithm of [39] (SC-FO) have been implemented in C++. The experiments have been carried out on an Intel(R)

Table 4: Verification Results for SP Multipliers

Benchmark	I/O bits	Commercial (h:m:s)	CPP [85] (h:m:s)	SC-FO [39] (h:m:s)	SC-LR (h:m:s)
SP-AR-RC	16/32	00:00:01	00:01:23	00:00:01	00:00:02
SP-WT-CL	16/32	00:00:01	00:00:46	TO	00:00:05
SP-RT-KS	16/32	00:00:43	-	TO	00:00:17
SP-CT-BK	16/32	00:00:59	00:00:43	TO	00:00:04
SP-AR-RC	32/64	00:00:11	02:34:40	00:00:09	00:00:21
SP-WT-CL	32/64	00:00:06	00:15:12	TO	00:03:27
SP-DT-HC	32/64	00:00:09	00:21:14	TO	00:02:05
SP-CT-BK	32/64	TO	00:21:20	TO	00:01:35
SP-AR-RC	64/128	00:02:52	94:37:20	00:02:56	00:07:40
SP-WL-CL	64/128	00:00:36	05:46:40	TO	02:18:34
SP-RT-KS	64/128	TO	-	TO	02:51:12
SP-CT-BK	64/128	TO	05:31:44	TO	00:47:48
SP-AR-RC	128/256	01:03:34	TO	00:48:03	02:08:51
SP-CT-BK	128/256	TO	78:11:12	TO	14:03:33

Core(TM) i5-3320M CPU (2.6 GHz, 16 GByte) running Linux. For the experiments, the multipliers are given as Verilog RTL code. The designs were synthesized to gate level netlists using *Yosys* [101].

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

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

In Tables 4 and 5, we compare the runtimes of the proposed technique SC-LR, against our re-implementation of SC-FO [39], the presented algorithm in Chapter 3 which is referred as *Checking Partial Product* (CPP) approach, and the equivalence checker of the commercial tool OneSpin (after enabling multiplier options). The first column of Tables 4 and 5 shows the name of the circuit. The second column gives the number of inputs and output bits. The next four

Table 5: Verification Results for BP Multipliers

Benchmark	I/O bits	Commercial (h:m:s)	CPP [85] (h:m:s)	SC-FO [39] (h:m:s)	SC-LR (h:m:s)
BP-AR-RC	16/32	00:00:14	-	TO	00:00:02
BP-WT-CL	16/32	00:00:16	-	TO	00:00:09
BP-RT-KS	16/32	00:00:18	-	TO	00:00:17
BP-CT-BK	16/32	00:00:13	-	TO	00:00:06
BP-AR-RC	32/64	TO	-	TO	00:00:17
BP-WT-CL	32/64	TO	-	TO	00:04:46
BP-RT-KS	32/64	TO	-	TO	00:05:36
BP-CT-BK	32/64	TO	-	TO	00:02:20
BP-AR-RC	64/128	TO	-	TO	00:05:06
BP-WT-CL	64/128	TO	-	TO	03:03:48
BP-DT-HC	64/128	TO	-	TO	00:58:44
BP-CT-BK	64/128	TO	-	TO	00:37:53
BP-AR-RC	128/256	TO	-	TO	01:29:10
BP-CT-BK	128/256	TO	-	TO	15:14:49

columns provide the runtimes. The time out (TO in the table) has been set to 100 hours. For the CPP approach, "-" refers to the fact that CPP cannot be used for some architectures of multipliers. The experimental results clearly demonstrate the advantage of the proposed enhancement. While for multipliers with simple partial products (Table 4) the other approaches sometimes can verify the correctness, for the complex parallel architectures (Table 5) only our enhanced technique solves the verification problem when the instances reach relevant sizes. As can be seen, we are able to verify the correctness for up to 128 bits.

Please note that all benchmarks time out after 100 hours when performing verification using a naive miter construction (one big miter; ABC [13] using command '*cec*').

Table 6 shows some statistics about the SC-LR. The columns give the circuit name, number of circuit bits, number of vanishing monomial that are canceled by one assignment rule (#CVM), the run-time of the IMT after model rewriting, and finally statistics on the rewritten model. The model statistics columns show

Table 6: Statistics for Verification of Multipliers by SC-LR

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

number of polynomials (#P), number of monomials (#M), maximum size of a polynomial wrt. its number of monomials (#MP), and maximum size of a monomial wrt. its number of variables (#VM). The results of Table 6 show that multipliers with carry look ahead adders or with Kogge-stone adder have the largest number of vanishing monomials and therefore the largest execution time. Furthermore, it can be seen that the SC-LR spent most of the execution time in rewriting the circuit model.

A further remarkable result is that for the 64-bit BP-WT-CL, the number of vanishing monomials is “325377”, while the size of its model after rewriting is “71473” monomials. This means that the number of these redundant terms (vanishing monomials) is four times larger than the size of the overall model, which explains the major influence of removing vanishing monomials on reducing the complexity of the verification problem.

4.8 SUMMARY AND FUTURE WORK

In this chapter, we have presented innovative ideas which enhance the symbolic computation technique for verification of complex parallel multiplier architectures.

First, we have analyzed the computational complexity of verifying multipliers described as SCNs, which is proved to be polynomial in space under a specific substitution order.

Then, a model rewriting algorithm has been proposed to lift bit-level representations of complex parallel multiplier into SCNs. The algorithm is based on canceling vanishing monomials that are distributed within multiplier models in an efficient way before their exponential blow-up during the IMT. It rewrites the algebraic model of the multiplier based on the XOR gates of the netlist in that it reveals monomials that satisfy the one assignment rule, making the removing of theses vanishing monomials very simple.

Also, the IMT has been qualified by new substitution rules to take better decisions, bypassing unnecessary exponential blow-up in space during recursive divisions performed by the IMT procedure.

Finally, experimental results have demonstrated the efficiency of the enhanced technique, i.e. for all complex parallel multipliers we verified the correctness within seconds to 15 hours (for 128 bits), while all other approaches reached the timed out limit of 100 hours and gave no result.

Directions for future work include:

1. Sophisticated approaches to extract exactly the XOR information from a netlist are motivated by the problem that the XOR rewriting scheme depends on structural information of the netlist like XORs which are not always available.

2. The IMT procedure can be upgraded by new heuristics for taking substitution decisions and exploiting successful ideas from the field of satisfiability solvers (SAT/SMT) such as effective learning approaches.

3. Heavy optimized multipliers are still major challenges for all formal verification techniques including the enhanced symbolic computation technique since SCNs cannot be constructed from such optimized netlists. Investigating this problem is a necessity for automated verification of circuits that incorporate such multipliers.

5

EQUIVALENCE CHECKING OF FLOATING-POINT MULTIPLIERS USING GRÖBNER BASES

Boolean reasoning based on Gröbner bases (available with symbolic computation packages) offers a robust mechanism that verifies arithmetic circuits at gate-level (see e.g., [39, 78, 87]) and their power has recently been demonstrated in formally verifying large class of bit-level multipliers, as shown in the previous chapter. Motivated by this recent success of the symbolic computation technique in formal verification of large scale gate-level multipliers, in this chapter, we propose an algebraic equivalence checking for handling circuits that contain both complex arithmetic components as well as control logic. These circuits pose major challenges for existing proof techniques including symbolic computation based reasoning techniques and no satisfactory solution has yet been presented. Toward solving this problem, we propose *Algebraic Combinational Equivalence Checking* (ACEC), a technique that allows reasoning over circuits which combine data-path and control logic using symbolic computation. To the best of our knowledge, this is the first full automated technique to formally verify binary floating-point circuits without any kind of case splitting or other manual efforts.

A naive algebraic equivalence checking to verify such circuits models the two compared circuits in the form of Gröbner bases over the Boolean ring and combines them into a single algebraic model. Then it checks the equivalences between the corresponding outputs of the two circuits by testing their membership in the combined model, as shown in Figure 22. The problem of this setting is that during the recursive divisions performed by the ideal membership testing (IMT) the sizes of the resulted remainders blow up exponentially. Since the IMT does not scale for the described setting, we propose *reverse engineering* to identify boundaries of arithmetic components and to abstract them to canonical representations. Further, we propose *arithmetic sweeping* which utilizes the abstracted components to find and prove internal bit and word equivalences between both circuits. The ACEC integrates the reverse engineering and the arithmetic sweeping algorithms to decompose automatically the verification problem, which bypasses the exponential blow-up of the problem size. We demonstrate the applicability of ACEC for checking the equivalence of a floating point multiplier (including full IEEE-754 rounding scheme) against

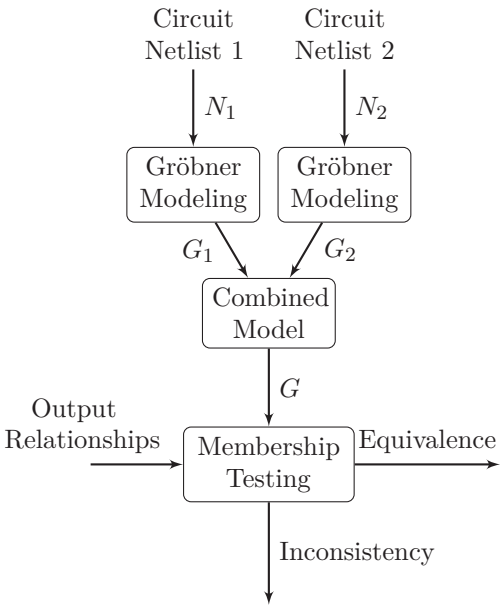

Figure 22: Naive Equivalence Checking Setting

several optimized and diversified implementations which cannot be verified by other proof techniques.

5.1 ALGEBRAIC COMBINATIONAL EQUIVALENCE CHECKING

Given two circuits C_1 and C_2 that represent the functions $f_1(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$ and $f_2(x_1, \ldots, x_n) = (z_1, \ldots, z_m)$, respectively, our aim is to show equivalence of C_1 and C_2, i.e., $(y_1, \ldots, y_m) = (z_1, \ldots, z_m)$ for all $x_1, \ldots, x_n$. We propose to solve the problem using algebraic computation methods. Since the specification of C_1 and C_2 may be unknown or since it may not be expressible in a canonical and an abstract form over $\mathbb{Z}_{2^n}$, we cannot use previous work [39, 78, 87] that performs ideal membership testing with respect to a given specification.

Instead we propose to represent C_1 and C_2 as polynomial sets G_1 and G_2 and combine them into a single model $G = G_1 \cup G_2$. We then formulate the problem as testing the membership of relations between variables in C_1 and C_2 wrt. G. An obvious choice for such a relation is the equivalence of output signals $y_i = z_i$ which can be expressed in a polynomial as $y_i - z_i = 0$. However, reducing such a polynomial wrt. G causes a tremendous overhead since the substitution of all the internal variables in G_1 and G_2 will blow up the sizes of the polynomials in G.

To overcome this problem we suggest to find internal equivalences, i.e., polynomials that express equivalence of two internal signals in G_1 and G_2. Reducing these polynomials wrt. G causes a smaller overhead and simplifies G. This technique is similar to SAT sweeping in combinational equivalence checking [66] and we call it arithmetic sweeping in the following. Arithmetic sweeping works as follows: for each internal variable v_1 in G_1 we search for an equivalent variable v_2 in G_2, i.e., v_1 and v_2 represent the same function wrt. to the primary inputs. We call such a pair (v_1, v_2) *bit equivalence* and are able to substitute v_2 by v_1 in all polynomials. For some internal variables, we will not be able to prove equivalence to another variable. These variables are eliminated by the substitution for proved bit equivalent variables of their transitive fan-in.

However, performing arithmetic sweeping on the overall combined model G is not scalable. First, the number of candidates for bit equivalences is too large, and second, checking for equivalence a pair of variables that have a large transitive fan-in may be too difficult. To circumvent this problem, we first apply *reverse engineering* for two main goals: i) extracting and abstracting arithmetic word-level components to canonical polynomials; ii) partitioning the circuits G_1 and G_2 into smaller parts. The algorithm works as follows: First, we try to find an instance of an arithmetic word-level component both in G_1 and G_2 and abstract them to canonical polynomials. If this is successful, we obtain an input boundary and an output boundary for the component in G_1 and G_2. The pairs of input boundaries and output boundaries are candidates for *word equivalences*. Having them, we perform arithmetic sweeping only in the transitive fan-in of the input boundaries. If this ultimately proves that the input boundaries are equivalent and we have proven that *abstracted polynomials* of the two arithmetic components found by reverse engineering are equivalent, we can merge the transitive fan-in of the output boundaries from G, making the model significantly smaller.

The overall flow of the ACEC is demonstrated by Figure 23. It starts by modeling the two compared netlists N_1 and N_2 and combining them in one Gröbner basis model G over the Boolean ring. Then it performs successively the two main algorithms of the ACEC, which are reverse engineering and arithmetic sweeping, ensuing a simplified model of G named G_{simple}. Finally, as shown in the end of Figure 23, the ACEC tests the consistency between output relationships and the simplified model G_{simple}. In the middle of Figure 23 the reverse engineering algorithm is presented, it rewrites the model G to lift concealed bit-level arithmetic components into sum carry networks (SCNs), afterwards it exploits features of SCNs to identify boundaries of arithmetic components in the rewritten Gröbner basis model G'. The last task of reverse engineering is

the abstraction of founded SCNs into word-level polynomials using *Gaussian elimination*, storing them in a word-level model G_{word}. After applying reverse engineering, the arithmetic sweeping algorithm is evoked, as shown in Figure 23, the arithmetic sweeping leverages the obtained polynomials of G_{word} and G' to deduce and prove equivalence relationships between internal variables of G', which leads to the simplified model G_{simple} by merging internal variables of G' that are proved to be equivalent.

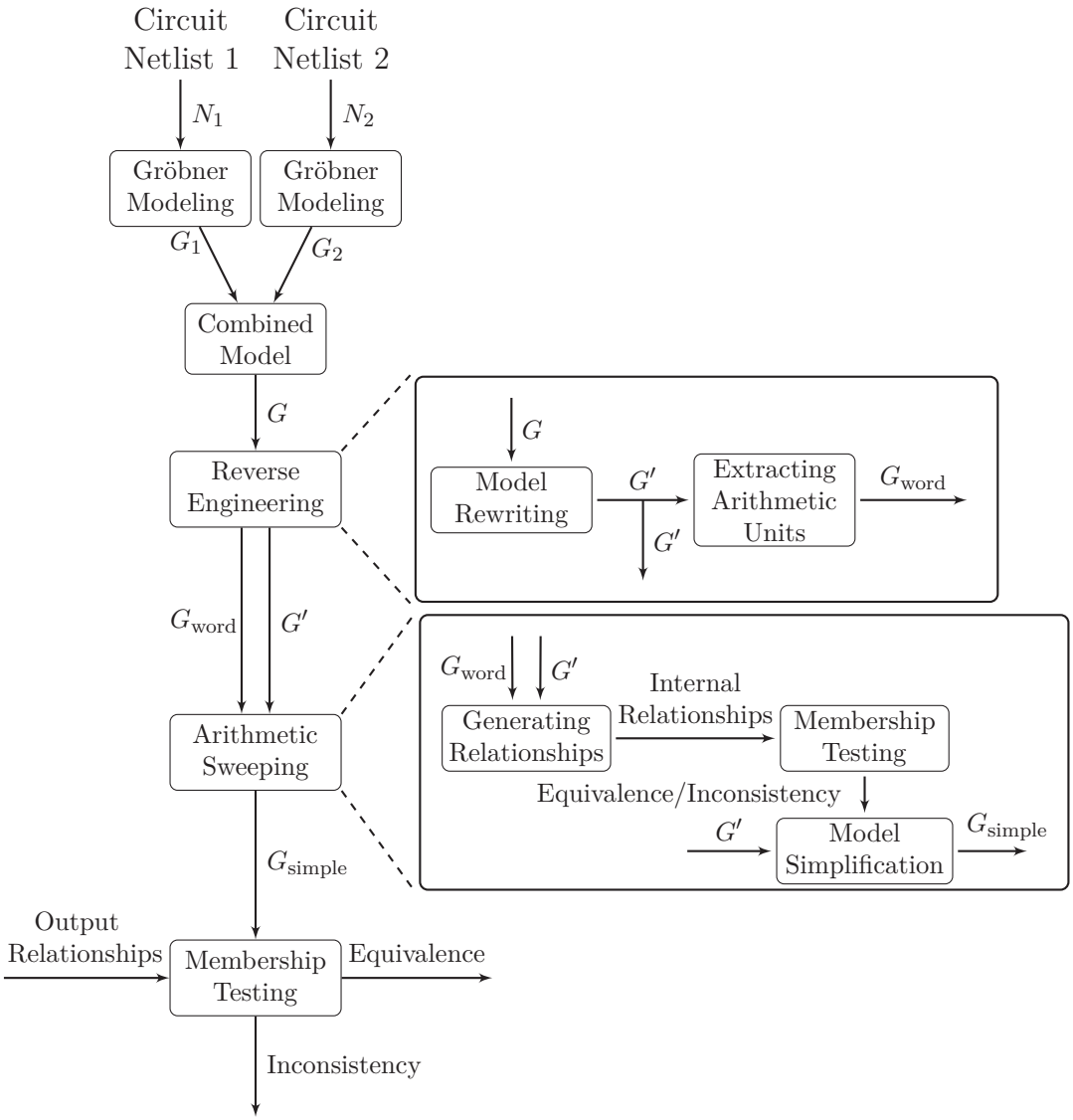

Figure 23: Flow of ACEC

Details on the algorithms of our ACEC are explained in the remaining sections. To summarize, the main contributions of this chapter are:

1. Utilizing symbolic computation for the combinational equivalence checking of bit-level circuits.
2. Identifying boundaries of arithmetic components that exist within a larger bit-level circuit together with abstracting their functions to canonical word-level descriptions using a new reverse engineering algorithm.
3. Partitioning the circuits into smaller parts based on boundaries of the extracted word-level components.
4. Proposing arithmetic sweeping which leverages the given arithmetic information to find (and prove) internal equivalences.
5. Offering efficient polynomial representations for the control logic functions based on functional decomposition.

5.2 REVERSE ENGINEERING OF DATA-PATH UNITS

Key in ACEC is to find arithmetic components using the reverse engineering in order to guide the arithmetic sweeping in decomposing the verification problem. Reverse engineering extracts data-path components and abstracts them to canonical word-level polynomials. To locate such units, the propagation of carry bits between internal nets of data-path units is a property that can be used to identify them. In the proposed reverse engineering algorithm we exploit this property to extract data-path units from the combined model G. According to our observation, these carry bits are modeled as carry terms (see Definition 22) distributed among polynomials of G. The carry terms are nonlinear terms that can be distinguished by their shared monomials and their coefficients which are multiple of each other and with opposite signs. However, carry terms are hidden within bit-level models, they are exposed only when bit-level descriptions of arithmetic components are rewritten into *Sum Carry Networks* (SCNs, see Definition 21). Such rewriting is performed by a model rewriting algorithm based on the principles explained in Section 4.4. The algorithm is applied on the combined model G to describe arithmetic functions as SCNs, revealing carry terms among polynomials of adder cells of SCNs. Having a rewritten model G', the reverse engineering algorithm utilizes the feature of carry terms to identify

the boundaries of each SCN in G', isolating each set of polynomials that represent a SCN. The final step of the reverse engineering is to derive *one* canonical polynomial for each identified SCN using Gaussian elimination.

In contrast to this proposed algorithm, the recently reverse engineering algorithms [94, 103] for the extraction of arithmetic word level components from a gate-level netlists are not applicable to designs with a non-arithmetic combinational logic attached to the output.

In the following, we introduce the model rewriting algorithm in Subsection 5.2.1, then the two main tasks of the reverse engineering algorithm to identify and abstract data-path units are provided in Subsections 5.2.2 and 5.2.3.

5.2.1 *Model Rewriting*

The proposed reverse engineering algorithm leverages the model rewriting algorithm (see Section 4.4) to expose a specific feature that distinguishes arithmetic functions than other control logic function when these functions are combined and implemented as a see of logic gates (netlist). Model rewriting has the ability to reveal carry terms among polynomials that model data-path units. This revealing for carry terms enables our reverse engineering algorithm to identify boundaries of different architectures of large scale multipliers and adders hidden in a given netlist. The rewriting algorithm executes successively two rewriting schemes named XOR rewriting and common rewriting. The first scheme XOR rewriting combines the knowledge of the circuit gates with the algebraic model. It rewrites the model such that it depends only on inputs and output variables of chains of XOR gates, whereas all other variables are substituted. The second common rewriting scheme rewrites the model obtained from XOR rewriting such that the model depends only on variables that are used in more than one polynomial. As presented in Section 4.4, the rewriting algorithm is capable of lifting bit-level models of a large class of arithmetic architectures into SCNs which reveal carry terms, making it feasible for the reverse engineering algorithm to identify polynomial sets of each SCN modeling a data-path unit.

However, the rewriting algorithm has been designed under the assumption that its input is a bit level description of an arithmetic circuit. Applying this algorithm to a control logic circuitry causes a blow-up in the number of model terms since control logic usually does not contain XOR gates, consequently, most of the control logic variables are substituted. To take advantage of the rewriting algorithm for circuits which contain data-path and control logic, we

distinguish the control logic part of a circuit by its multiplexers (MUXes) and disallow XOR rewriting and common rewriting from substituting input and output variables of MUXes. This guarantees that both schemes will be applied only to the data-path logic. For a given bit-level model G which consists of control and data-path logics, the modified rewriting algorithm generates a new model G' that has a set of polynomials describing functions of chains of XORs, MUXes, and cones of gates which are bounded by inputs and outputs of XORs and MUXes. Thus the reverse engineering algorithm obtains a rewritten model G' that describes data-path components as SCNs and in the same time polynomials of control logic have no exponential size.

5.2.2 *Identifying Boundaries of Data-path Units*

Obtaining the rewritten model G', the next task of the reverse engineering algorithm is to identify all SCNs that lie within G' using the feature of carry terms. Carry terms appear only among polynomials of adder cells which are connected by their sum and carry variables to build SCNs. Based on this structure, the algorithm identifies sets of polynomials that represent SCNs as follows:

1. First, the algorithm groups polynomials of G' that share carry terms into sets, such that a new polynomial joins a set aG_i that models an adder cell i, if it shares a carry term with other polynomials in the set aG_i, e.g, the polynomials g_0 and g_1 are in the same adder cell set, if one of them has the term $-x_0x_1$ while the second has the term $2x_0x_1$.

2. Second, after building sets of adder cells aG_i, the algorithm inserts those that are connected to each other in a larger set nG_j representing a SCN. To perform this task, it relates each adder cell aG_i to a set of input variables X_i and a set of output variables Xo_i, which represent the inputs and the outputs of the adder cell, exploiting that a variable of a leading monomial of a Gröbner basis polynomial is the output of the Boolean function modeled by this polynomial. Then it builds nG_j based on the rule that two sets $\{aG_i, aG_{i+1}\} \in nG_j$, if $X_{i+1} \cap Xo_i \neq \{\}$. In other words, if some outputs of the adder cell aG_i are given as inputs to the adder aG_{i+1}.

Example 18. *To illustrate the way of identifying SCNs, consider the case of a 3-bit ripple carry adder that its polynomials are :*

$g_6 := -c_2 \boxed{-2c_1y_2x_2 + c_1y_2 + c_1x_2 + y_2x_2}$

$g_5 := -s_2 \boxed{+4c_1y_2x_2 - 2c_1y_2 - 2c_1x_2 - 2y_2x_2} + c_1 + y_2 + x_2$

$g_4 := -c_1 \boxed{-2c_0y_1x_1 + c_0y_1 + c_0x_1 + y_1x_1}$
$g_3 := -s_1 \boxed{+4c_0y_1x_1 - 2c_0y_1 - 2c_0x_1 - 2y_1x_1} + c_0 + y_1 + x_1$
$g_2 := -c_0 \boxed{+y_0x_0}$
$g_1 := -s_0 \boxed{-2y_0x_0} + y_0 + x_0$
To define this set of polynomials as a SCN, the reverse engineering algorithm assigns first polynomials that share carry terms into sets of adder cells. It deduces simply three sets which are $aG_1 = \{g_1, g_2\}$, $aG_2 = \{g_3, g_4\}$, *and* $aG_3 = \{g_5, g_6\}$. *Second, it relates each adder cell with its inputs and outputs. This means that* aG_1 *is related to* $X_1 = \{x_0, y_0\}$ *and* $Xo_1 = \{c_0, s_0\}$, aG_2 *has the sets* $X_2 = \{x_1, y_1, c_0\}$ *and* $Xo_2 = \{c_1, s_1\}$, *and* aG_3 *has the sets* $X_3 = \{x_2, y_2, c_1\}$ *and* $Xo_3 = \{c_2, s_2\}$. *Since* $X_2 \cap Xo_1 = \{c_0\}$ *and* $X_3 \cap Xo_2 = \{c_1\}$, *the set of polynomials that builds the SCN is* $nG = aG_1 \cup aG_2 \cup aG_3 = \{g_1, g_2, g_3, g_4, g_5, g_6\}$.

In addition to collecting polynomials of each SCN in a set nG, the reverse engineering algorithm determines inputs and outputs boundaries of these extracted SCNs. This works as follows: The algorithm extracts this information from inputs X_i and outputs Xo_i related to adder cells sets $aG_i \in nG$. A variable x in Xo_i that is not used as inputs for other adder cells ($x \in Xo_i$ and $x \notin X_k$ for all $k > i$) is identified as an output variable of this SCN and is stored in the set of variables nZ which represents the outputs boundary. Similarly, a variable x is inserted in the set nX of the inputs boundary, if $x \in X_i$ and $x \notin Xo_k$ for all $k < i$.

So far, we have presented the first task of the reverse engineering algorithm to identify boundaries of data-path units by building sets of extracted SCNs. Another task of the algorithm is presented in the following subsection, which is handling each SCN as an independent algebraic ideal to abstract it to one canonical polynomial using Gaussian elimination.

5.2.3 *Abstracting Data-path Units*

Expressing a function by a canonical representation allows checking the equivalence between two different implementations of the function in a linear time. This why the abstraction of data-path units to canonical polynomials is crucial for the ACEC. The proposed reverse engineering performs this task on the extracted SCNs, exploiting a feature of SCN that it can be reduced without exponential complexity in space. Lemma 6 and Lemma 8 have proved that the reduction of SCNs wrt. specification polynomials is bounded by linear complexity for adders and by polynomial complexity in the case of multipliers under a specific substitution order. The reverse engineering algorithm deploys Gaussian

elimination to reduce SCNs into what are called reduced Gröbner bases (see Definition 18) which are canonical polynomials for SCNs. The reduction of SCNs by Gaussian elimination has a computational complexity similar to Lemma 6 and Lemma 8, on the other hand, it does not require a specification polynomial. The idea of leveraging Gaussian elimination algorithm has been proposed by [35, 40] to replace Buchberger's algorithm [19] for computing Gröbner bases. In the context of reverse engineering, Gaussian elimination is utilized for another application which is the reduction of a given SCN into one canonical polynomial. For this propose, Gaussian elimination algorithm has been modified in the sense that it performs only this specific task. The modified version is called *Gaussian Network Reduction* (GNR). It performs iteratively three steps:

1. Select from the given set nG of a SCN two polynomials g and $\hat{g}$ that have at least two terms t and $\hat{t}$ respectively which: 1) have the same monomials, and 2) the absolute values of their coefficients c_t and $c_{\hat{t}}$ are equal or multiple of each other. The highest priority for the selection is given to the pair of polynomials that have the largest number of such terms.

2. Let $\|c_t\| > \|c_{\hat{t}}\|$, calculate a scalar $c_q = -c_t/c_{\hat{t}}$, multiply $\hat{g}$ by c_q, and then add the result to g. This leads to cancel the terms t as well as $\hat{t}$ and therefore derives a new polynomial h.

3. Update the set nG by removing polynomials g and $\hat{g}$ together with inserting the new polynomial h, i.e., $nG = (nG \setminus \{g, \hat{g}\}) \cup h$. The GNR algorithm terminates when nG consists only of one polynomial.

Example 19. *To illustrate the GNR algorithm, consider again the set nG of a 3-bit ripple carry adder:*

$g_6 := -c_2\,\boxed{-2c_1y_2x_2 + c_1y_2 + c_1x_2 + y_2x_2}$

$g_5 := -s_2\,\boxed{+4c_1y_2x_2 - 2c_1y_2 - 2c_1x_2 - 2y_2x_2} + c_1 + y_2 + x_2$

$g_4 := -c_1\,\boxed{-2c_0y_1x_1 + c_0y_1 + c_0x_1 + y_1x_1}$

$g_3 := -s_1\,\boxed{+4c_0y_1x_1 - 2c_0y_1 - 2c_0x_1 - 2y_1x_1} + c_0 + y_1 + x_1$

$g_2 := -c_0\,\boxed{+y_0x_0}$

$g_1 := -s_0\,\boxed{-2y_0x_0} + y_0 + x_0$

The given SCN model consists of polynomials which have common monomials such as g_6, g_5 (colored green/dashed box in the example). The similar structural property can be seen for equally colored terms of the polynomials g_4, g_3 and polynomials g_2, g_1, respectively. The GNR algorithm selects first the polynomials g_6 and g_5. In order to cancel their common terms, it multiples g_6 by 2 and adds

it to g_5. The result is the polynomial $h_1 := -2c_2 - s_2 + c_1 + y_2 + x_2$ which represents a full adder function (step two of the algorithm). The third step is to update the set nG to be

$h_1 := -2c_2 - s_2 + c_1 + y_2 + x_2$
$g_4 := -c_1 \boxed{-2c_0y_1x_1 + c_0y_1 + c_0x_1 + y_1x_1}$
$g_3 := -s_1 \boxed{+4c_0y_1x_1 - 2c_0y_1 - 2c_0x_1 - 2y_1x_1} + c_0 + y_1 + x_1$
$g_2 := -c_0 \boxed{+y_0x_0}$
$g_1 := -s_0 \boxed{-2y_0x_0} + y_0 + x_0$

Applying the same steps on other related polynomials yields another two polynomials of full adders $h_2 := -2c_1 - s_1 + c_0 + y_1 + x_1$ and $h_3 := -2c_0 - s_0 + y_0 + x_0$, resulting in the updated nG:

$h_1 := -2c_2 - s_2 + c_1 + y_2 + x_2$
$h_2 := -2c_1 - s_1 + c_0 + y_1 + x_1$
$h_3 := -2c_0 - s_0 + y_0 + x_0$

Performing the GNR steps again on the three full adder polynomials cancels shared terms and achieves a reduced Gröbner basis. The algorithm multiplies h_1 by 2 and adds h_2. The result will be $h_4 := -4c_2 - 2s_2 - s_1 + 2y_2 + 2x_2 + c_0 + y_1 + x_1$. Finally, the reduced Gröbner basis polynomial $h_5 := -8c_2 - 4s_2 - 2s_1 - s_0 + 4y_1 + 4x_1 + 2y_1 + 2x_1 + y_0 + x_0$ is derived by multiplying h_4 by 2 and adding it to h_3 for canceling the shared monomial c_0.

Three questions could be asked about this proposed algorithm: Does the GNR produce always one polynomial?. Is the resulted polynomial a canonical representation for a given set of a SCN?. What is the computational complexity of the GNR algorithm?. In the following, we answer these questions by three lemmas.

Lemma 9. *Let $nG = \{g_1, \cdots, g_m\}$ denotes the polynomial set of a SCN given to the GNR, the algorithm derives always one polynomial from nG.*

Proof. The construction of nG guarantees that each subset of polynomials $aG_i \subset nG$ shares at least one monomial with other subset of polynomials $aG_j \subset nG$ regardless of the sizes of these subsets (see Subsection 5.2.2). Besides this, the GNR algorithm works iteratively on reducing each pair of related polynomials (that share monomials) into one polynomial. Because of the construction of nG, there will be always during the process of the algorithm a pair of related polynomials h_i and h_j that can be reduced to one polynomial. h_i and h_j are derived from reducing subsets aG_i and aG_j that have at least one shared monomial, respectively, these shared monomials will remain between h_i and h_j, thereby the GNR can replace both of them by a new polynomial. By re-

placing all related pairs of polynomials, the GNR terminates always by a single polynomial. ■

Lemma 10. *Let a set nG that has been reduced by the GNR algorithm to a single polynomial $n\hat{G} = \{g\}$, g is the unique canonical representation of the function f modeled by the nG, wrt. a monomial order $\prec$.*

Proof. Based on Lemma 2, for every ideal there is a unique reduced Gröbner basis. The GNR has reduced the set of polynomials nG (ideal) to only one polynomial $n\hat{G} = \{g\}$. Since $n\hat{G} = \{g\}$ has a single polynomial, no term in g is divisible by the leading term of any other polynomial in $n\hat{G}$. This means that Definition 18 of the reduced Gröbner basis holds for $n\hat{G}$, therefore, g is a canonical abstracted representation under the monomial order $\prec$ for the function f implemented by the set nG. ■

Lemma 11. *Let n_t denotes the total number of terms of all polynomials in the set $nG = g_1, \cdots, g_m$ that represents a SCN of an integer adder or a SCN of an integer multiplier, and let $Z = \{z_0, \cdots, z_m\}$, $X = \{x_0, \cdots, x_{n-1}\}$ and $Y = \{y_0, \cdots, y_{n-1}\}$ represent outputs and inputs of the function modeled by the SCN, respectively. The reduction of nG by the GNR algorithm is bounded in space by a linear complexity for adders and by a polynomial complexity in case of multipliers.*

Proof. Since the GNR algorithm reduces after each iterative step by at least two terms from the total number of terms n_t, the complexity of each step in the space is always bounded by $\mathcal{O}(n_t)$. This means that the size of each resulted polynomial will not exceeds n_t regardless of the type of the given set of polynomials. In case of integer adders or integer multipliers modeled as SCNs, their set of polynomials reveal all carry terms which are canceled simply by the GNR algorithm. Therefore, the n-bit addition function is reduced into one polynomial $h := -\sum_{k=0}^{n} 2^k z_k + \sum_{k=0}^{n-1} 2^k x_k + \sum_{k=0}^{n-1} 2^k y_k$ whose size is bounded by $\mathcal{O}(n)$, while for the n-bit multiplication the resulted polynomial $h := -\sum_{k=0}^{2n-1} 2^k z_k + \sum_{k=0}^{n-1} 2^k x_k \cdot \sum_{k=0}^{n-1} 2^k y_k$ is of the size bounded by $\mathcal{O}(n^2)$. ■

The canonical polynomials for each extracted arithmetic components are stored in a set of word polynomials G_{word} after relating each polynomial with a set of its input variables nX as well as a set of its output variables nZ. In the next section, we present how the arithmetic sweeping algorithm deploys the extracted arithmetic information to simplify the rewritten model G'.

5.3 ARITHMETIC SWEEPING

Arithmetic sweeping aims to find internal equivalences, which avoids prohibitive runtime during recursive divisions of the IMT. Of course, when having identified candidates for internal equivalence, it is still necessary to prove their equivalence (which is also done using the same IMT procedure). Hence, to gain an overall benefit, we need i) promising candidates and ii) moderate runtimes for the equivalence proofs. Our proposed arithmetic sweeping reaches both goals as follows.

For i), the reverse engineering step provides arithmetic components. From this, we generate promising candidates based on the I/O boundaries of these components. The algorithm uses the I/O boundaries to partition the variables of the combined model G' into groups. Simulation deduces *word equivalence* (wE; for details see below) candidates between outputs of the arithmetic components. For every nominated wE the partitioning of model variables is performed by classifying two groups of variables. One for the transitive fan-ins variables of the input boundaries of wE and the other are internal variables of the two related arithmetic components. Deducing only internal *bit equivalences* (bE; see below) between variables in the same group increases the potential of equivalence.

For ii), the equivalence proofs become feasible for several reasons. Arithmetic sweeping generates two types of relationships which are *bit equivalence* (bE) pair and *word equivalence* (wE) pair. bE describes the equivalence of a pair of variables $(v,\hat{v})$ and is formulated by the polynomial $g := -v + \hat{v}$. The word equivalence (wE) polynomial is formulated as $g := Z - \hat{Z}$ for the word pair candidate $(Z,\hat{Z})$, where $Z = 2^m z_m + \cdots + z_0$ and $\hat{Z} = 2^m \hat{z}_m + \cdots + \hat{z}_0$. For each arithmetic component we have determined an *abstract* canonical polynomial in the reverse engineering step. The major advantage over SAT sweeping is that the proof for the internal equivalences is performed by dividing wE polynomials wrt. the abstracted polynomials. For doing this, the word model G_{word} obtained from the reverse engineering algorithm is modified as follows: For every abstracted polynomial $-2^m z_m - \cdots - z_0 + f(x_0, \cdots, x_n)$, an integer word variable Z is created and the polynomial is replaced by $-Z + f(x_0, \cdots, x_n)$. The polynomial $-Z + 2^m z_m + \cdots + z_0$ is used to interpret the equivalence between two output words Z and $\hat{Z}$, as shown in Lemma 12 of Section 5.3.2. To summarize, dividing wE wrt. abstracted polynomials has a major influence on the performance of the technique—it avoids the exhaustive cost of searching for equivalences between internal variables of the data-path units which usually have the large number of non-equivalent variables in their transitive fan-ins.

5.3.1 *Generating Relationship Polynomials*

The choice of relationship candidates is always the main problem of different equivalence checking techniques. ACEC draws on the simulation approach of [43] and the extracted data-path polynomials to deduce bit and word relationships. Four steps are performed to generate relationship polynomials i) nominating wE polynomials, ii) classifying the model variables to groups, iii) generating bE polynomials, and finally iv) sorting wE and bE polynomials in a relationship list.

Based on a fixed size of global simulation over the primary inputs of G', word relationships between the output words of the data-path polynomials are deduced. Two words build a wE polynomial if their integer values are equal under all simulated assignments.

The approach classifies the variables of G' to groups according to wE polynomials. One wE polynomial categorizes two groups, the first consists of all transitive fan-in variables of the polynomial; and the second contains internal variables which are bounded by outputs and inputs variables of wE.

Example 20. *To illustrate this idea, consider a model which has four extracted data-path units (*DPU_1, DPU_2, DPU_3, *and* DPU_4*), as shown in Figure 24. The simulation nominates two wEs, one relates the output word of* DPU_1 *and* DPU_2*, the other one is between* DPU_3 *and* DPU_4*. The approach classifies the model variables into 5 groups i) a group for transitive fan-in variables of* DPU_1 *and* DPU_2*, ii) a group which contains internal variables of* DPU_1 *and* DPU_2*, iii) transitive fan-in variables of the wE between* DPU_3 *and* DPU_4*, iv) their internal variables, and v) the remaining variables of* C_1 *and* C_2 *which are not classified into groups.*

Classified groups of G' and global simulation are used to determine for every model variable v_i a set of variables ϕ_i. We have $v_j \in \phi_i$, if Boolean values of v_i and v_j are the same under each of input assignments; and therefore v_i and v_j belong to the same classified group. Finally, bE polynomials between v_i and other variables of ϕ_i are generated. We call them *bE polynomials* of the variable v_i.

After classifying model variables and generating wE and bE relationships, these nominated relationships are sorted topologically wrt. the circuit and their leading variables. The sorting procedure aims to test a wE polynomial after testing all bE polynomials of variables in its transitive fan-in group. First, the wE polynomials are sorted topologically. Next, the procedure iterates over the wE polynomials for inserting in the list for every wE i) bE polynomials of variables

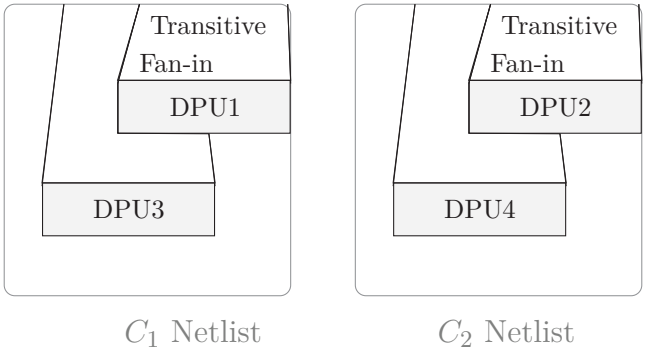

Figure 24: Schematic of a Model with Word Relationships

in the transitive fan-in group of this wE, ii) the wE polynomial itself, then iii) bE polynomials of variables in its internal group. Finally, the bE polynomials of remaining variables that are not included in groups that are related to wEs, are inserted at the end of the list.

5.3.2 *Testing Membership of Internal Relations*

During the testing of internal relationships, the IMT algorithm is invoked to divide every polynomial p_r from the relationship list wrt. G' or G_{word}, if p_r is a bE polynomial, the division is done wrt. G', otherwise is performed wrt. G_{word}. Based on the remainder result of dividing p_r, the approach eliminates or merges variables of p_r from the models G' and G_{word}.

The merging decision is taken, in the case that the remainder result of dividing p_r is equal to zero. The approach merges every two variables of p_r which are proved to be functionally equivalent to one variable. In case that p_r is a wE polynomial, equivalence is proved based on the following lemma.

Lemma 12. *Given the equivalent of two integer words* $Z = 2^m z_m + \cdots + z_0$ *and* $\hat{Z} = 2^m \hat{z}_m + \cdots + \hat{z}_0$. *If* Z *and* $\hat{Z}$ *have the same number system and the number system is not redundant, then the bit variables* z_i *and* $\hat{z}_i$ *which have same weights (coefficients) are equivalent.*

Merging results are reducing the number of polynomials in G', these merged variables are considered new primary inputs, therefore polynomials of their transitive fan-in variables are removed from G'.

Example 21. *Continue with the previous model example. According to the sorted relationship list, the first group to be tested is the bE polynomials between*

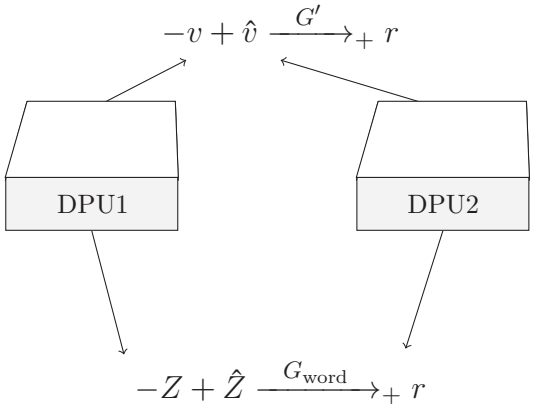

Figure 25: Bit and Word Relationships Testing

corresponding fan-in variables of DPU_1 *and* DPU_2*, and then the wE polynomial that describes an equivalence between the output word variables of* DPU_1 *and* DPU_2*, as shown in Figure 25. In Figure 26, a simplified model is appeared after merging variables that are proved to be equivalent by testing bE and wE polynomials of* DPU_1 *and* DPU_2*. The simplification for the original model is performed by removing polynomials that model* DPU_1 *and* DPU_2 *as well as those which model their transitive fan-in variables. In order to avoid redundant divisions, the remaining bE polynomials which test the already merged variables will be removed from the relationship list.*

A variable of the model that has no functional equivalences is eliminated by substituting it with the leading terms of its polynomials which are functions in proved bit equivalent variables. The elimination decision will be taken for variables v_i of p_r. If the remainder of dividing p_r is not equal to zero, and there are no more untested bE or wE polynomials in the list which are related to v_i. These eliminations facilitate the division process of next relationships. It increases the number of shared input variables among polynomials of G' which simplifies the division process of p_r wrt. G'. For example, dividing $p_r : -v_i + v_j$ wrt. a model that has polynomials $g_1 : -v_i + x_1x_2 + x_3$ and $g_2 : -v_j + x_1x_2 + x_3$ will be simplified to a subtraction operation. The remainder of the division will be $x_1x_2 + x_3 - x_1x_2 - x_3 = 0$.

5.4 EFFICIENT POLYNOMIAL REPRESENTATION

The polynomial is the heart of the algebraic computation technique. An efficient representation of a polynomial has a major impact on the performance

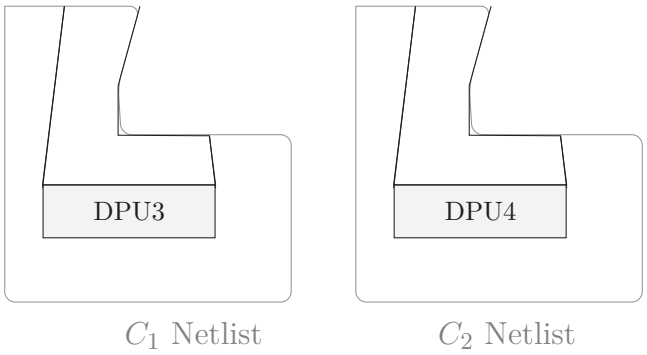

Figure 26: Schematic of a Simplified Model

of any algebraic algorithm. So far, only arithmetic functions have been implemented by canonical and compact representations, but what about other Boolean functions in the model G'. Unfortunately, some Boolean functions cannot be implemented by multivariate polynomials with moderate sizes. To circumvent the exponential sizes of the polynomials representing these Boolean functions, we propose a decomposition method which reduces the number of terms of polynomials significantly. The decomposition method offers also semi-canonical representations which makes it feasible to check the equivalence between two different implementations of the same Boolean function.

5.4.1 *Different Decompositions*

Inspired by decomposition types of decision diagrams [24, 32], we enhance representations of polynomials by considering two decomposition types:

$$\begin{aligned} f &= f_{|x=0} + x(f_{|x=1} - f_{|x=0}) && \text{positive Davio (pD)} \\ f &= f_{|x=1} + (1-x)(f_{|x=0} - f_{|x=1}) && \text{negative Davio (nD),} \end{aligned}$$

where x denotes a Boolean variable, the functions f are combined with addition, subtraction, and multiplication operations.

Our observation is that polynomials have been typically represented by the pD decomposition. This obstructs a compact representation for some Boolean functions such as a chain of OR gates. Although, these functions can be canonically defined without an exponential size, if another type of decomposition for variables is considered such as nD.

Example 22. *Consider a 4-input OR function* $f(x_0, x_1, x_2, x_3)$*, its polynomial representation that follows only pD is* $f = x_0 + x_1 + x_2 + x_3 - x_0x_1 - x_0x_2 - x_0x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_0x_1x_2 + x_0x_1x_3 + x_0x_2x_3 + x_1x_2x_3 - x_0x_1x_2x_3$*. By decomposing* f *using nD for all of its variables, it will be* $f = 1 - \bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3$*, where* $\bar{x}_i = 1 - x_i$*. For an n-bit OR function, a polynomial which follows pD consists of* $2^n - 1$ *terms, while the nD polynomial has only two terms.*

Other Boolean functions that can be implemented efficiently using the combination between pD and nD decompositions are equality as well as inequality conditions with respect to constants, which can be illustrated by the following examples.

Example 23. *Consider the equality function* $f = (8x_3 + 4x_2 + 2x_1 + x_0 == 1000)$*, where* f *is equal to one when* $x_0 = 0$*,* $x_1 = 0$*,* $x_2 = 0$ *and* $x_3 = 1$*. The polynomial representation of the function that follows only pD is* $f = x_3 - x_3x_0 - x_3x_1 - x_3x_2 + x_3x_0x_1 + x_3x_0x_2 + x_3x_1x_2 - x_0x_1x_2x_3$*. By decomposing* f *using nD for variables* $\{x_0, x_1, x_2\}$ *and pD for the variable* x_3*,* f *is implemented as* $f = \bar{x}_0\bar{x}_1\bar{x}_2x_3$*. For an n-bit equality function, a polynomial with only one term can represent the function.*

Example 24. *Consider the inequality function* $f = (8x_3 + 4x_2 + 2x_1 + x_0 > 1000)$*, where* f *is equal to one when* $x_3 = 1$ *and* $x_0 = 1$*,* $x_1 = 1$ *or* $x_2 = 1$*. The polynomial representation of the function that follows only pD is* $f = x_3x_0 + x_3x_1 + x_3x_2 - x_3x_0x_1 - x_3x_0x_2 - x_3x_1x_2 + x_0x_1x_2x_3$*. By decomposing* f *using nD for variables* $\{x_0, x_1, x_2\}$ *and pD for the variable* x_3*,* f *is implemented as* $f = x_3 - \bar{x}_0\bar{x}_1\bar{x}_2x_3$*.*

Representing a Boolean function with a moderate size has a major influence on reducing the number of addition, subtraction, and multiplication operations, which enhances significantly the performance of any symbolic computation algorithm. In particular for the IMT procedure, most of the terms within the division process cancel each other before doing any further substitutions, in addition to reducing the computational cost of each division step since the number of terms involved in the process is less.

For applying different decompositions, we add to the Gröbner basis model negation version $\bar{v}_i$ for each variable v_i in the model, in addition to polynomials $g := -\bar{v}_i - v_i + 1$. As known from the field of decision diagrams, the choice of the type of the decomposition and the order of the variables plays a key role for the size of the diagram. In the context of multivariate polynomials, we fix the order of the variables to the reverse topological order and we propose

an approach to determine the *Decomposition Type* (DT) of each variable. As the main goal of applying different decompositions is reducing the number of polynomials terms, the decision of DT is taken based on this factor and the structure of the circuit.

For this purpose, we modify the modeling way of the circuit that is explained in Subsection 2.1.5 as follows:

$$\begin{aligned}
&\text{NOT:} && x_o = \neg x_1 && \Longrightarrow\ -x_o + \bar{x}_1\\
&\text{AND:} && x_o = x_1 \wedge x_2 && \Longrightarrow\ -x_o + x_1x_2\\
&\text{OR:} && x_o = x_1 \vee x_2 && \Longrightarrow\ -x_o + 1 - \bar{x}_1\bar{x}_2\\
&\text{XOR:} && x_o = x_1 \oplus x_2 && \Longrightarrow\ -x_o - 2x_1x_2 + x_1 + x_2\\
&\text{MUX:} && x_o = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3) && \Longrightarrow\ -x_o + x_1x_2 - x_1x_3 + x_3,
\end{aligned}$$

such that the DT of input variables of inverters and OR gates is nD, for AND, XOR, and MUX gates, it is pD. As shown in this modeling, one variable may have more than one DT in the model. During model rewriting, see Subsection 5.2.1, a polynomial g is rewritten by substituting one of its variables v_i, as a result of this step, another variable v_j in g may have different decomposition types—the variable v_j and its negation $\bar{v}_j$ are within the same polynomial g. In this case, we unify the DT of v_j based on the one which achieves the highest reduction with respect to the size of g. In case of n variables with different DTs within the same polynomial, the number of possible combinations of DTs for these variables are 2^n, e.g., for a polynomial with two variables v_1 and v_2, the possible decomposition combinations will be (v_1, v_2), $(v_1, \bar{v_2})$, $(\bar{v_1}, v_2)$, or $(\bar{v_1}, \bar{v_2})$. Trying all combinations to find the best representation leads to a prohibitive runtime because of calling the decomposition algorithm 2^n times. To bypass this problem, we propose a *polynomial decomposition approach* which takes the decomposition decision of every variable independently from others. This restriction on choosing DTs accelerates significantly the runtime of the proposed approach to find compact representations for polynomials. The polynomial decomposition approach works as follows,

1. During the model rewriting, its detects any rewritten polynomial g with one or more variables v_j that have non-unified DTs.

2. According to the reverse topological order, one variable v is selected. Given the two versions v and $\bar{v}$, the polynomial g is rewritten by replacing $\bar{v}$ with $1 - v$.

3. To find the proper DT for v, the decomposition algorithm (WLD) which designed for decision diagrams [31] is called. The input given to WLD

is the polynomial g after parsing it into a K*BMD, and it is asked to decompose the K*BMD of g wrt. v one time as pD and another time as nD. The resulted K*BMDs are parsed in the reverse direction into multivariate polynomials, the one with less size g_v is picked out and updates the original polynomial $g = g_v$.

4. Steps two and three are repeated iteratively until all variables of g have unified decomposition types.

Yet, we have shown that variables of a polynomial can be decomposed wrt. pD or nD decomposition types, but why Shannon (S) decomposition $f = (1-x)f_{|x=0} + xf_{|x=1}$ has not been also applied on a multivariate polynomial, although it is useful for implementing Boolean functions such as those composed of XORs or MUXes. The reason behind excluding Shannon decomposition is that it changes the representation of the XOR function from $-x_o - 2x_1x_2 + x_1 + x_2$ to be $-x_o + \bar{x}_1x_2 + x_1\bar{x}_2$. The problem of this description is that it hides the nonlinear term $-2x_1x_2$, therefore, carry terms of adder cells will be hidden since they are nonlinear terms by definition. The unrevealing of these carry terms will disable capabilities of the reverse engineering algorithm (see Section 5.2), it will not be able to extract arithmetic functions. This means that on one hand, Shannon decomposition is useful for representing functions such as shifters and comparators which are rich with XORs and MUXes, on the other hand, it cannot be applied on polynomials modeling integer-valued arithmetic functions. To cope with this problem, we have designed a modified version of the polynomial decomposition approach which permits Shannon decomposition, however, this modified version is enabled only after the reverse engineering algorithm. The modified polynomial decomposition is applied during the testing phase of the arithmetic sweeping algorithm (see Subsection 5.3.2). During this phase, variables that have no equivalences are eliminated from the model G', resulting in polynomials that may have variables with non-unified DTs. The variables of these polynomials are decomposed wrt. one of the three decomposition types: Shannon, nD or pD, using the modified polynomial decomposition approach.

The question now is whether this solution allows to represent shifters and comparators efficiently. This is can be explained by considering the model rewriting algorithm which prohibits the elimination of variables that model the outputs of XORs and MUXes. In the rewritten model G', the Boolean output functions of shifters and comparators are described by sets of polynomials modeling the compounded XORs and MUXes of these functions. With this status, G' is given

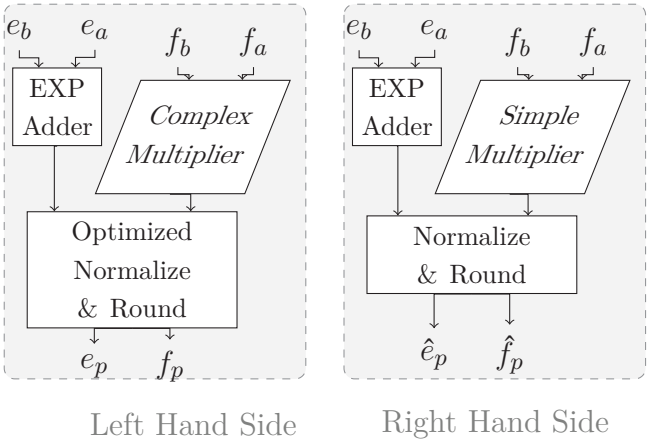

Figure 27: Compared FP Multiplier Circuits

to the arithmetic sweeping algorithm which tries to find equivalences between internal variables of shifters and comparators. In the case that outputs and internal variables of such functions are proved to have equivalences, then their descriptions in the model G' as polynomials of XORs and MUXes are sufficient to simplify the model, otherwise the arithmetic sweeping eliminates variables that have no equivalences. At this moment, the modified polynomial decomposition approach plays its role in providing polynomials with moderate sizes that model shifters and comparators after eliminating non-equivalence variables.

5.5 EXPERIMENTAL RESULTS

ACEC is implemented in C++. We compared it to the equivalence checkers of ABC [13] tool and a commercial tool (OneSpin). The experiments were carried out on an Intel(R) Core(TM) i5-3320M CPU (2.6 GHz, 16 GByte) running Linux.

We applied the ACEC to the problem of verifying floating-point (FP) multiplier. We have scaled and modified the structure of the FP multiplier unit of the open cores design module DOUBLE-FPU [67] for building dissimilar FP instances. As shown in Figure 27, the compared circuits have different multiplier architectures and their control logic units are optimized distinctively. The multiplier units are generated using the online tool *Arithmetic Module Generator* [2]. These generated circuits were synthesized from Verilog to gate level netlists using *Yosys* [101].

Table 7: Runtimes for Checking FP Multipliers Equivalences

Multiplier Architecture	FP operand # bits	Commercial # bits	ABC (h:m:s)	ACEC (h:m:s)	(h:m:s)
SP-CT-BK	16	00:08:50	TO	00:01:42	
SP-WT-CH	16	00:09:08	TO	00:01:44	
SP-CT-BK	24	TO	TO	00:17:49	
SP-WT-CH	24	TO	TO	00:25:58	
SP-CT-BK	32	TO	TO	02:24:01	
SP-WT-CH	32	TO	TO	03:41:43	
SP-CT-BK	64	TO	TO	TO	
SP-WT-CH	64	TO	TO	TO	

In Table 7, we demonstrate the runtimes of checking the equivalences of divergent FP multipliers against the same circuit reference. The reference consists of simple multiplier (SP-AR-RC) and unoptimized normalize round unit. While the compared circuits contain complex multipliers and round units which are optimized using the Yosys option (share[1]). The first column of Table 7 shows the type of the multiplier architecture. The second and the third columns give number of bits of an FP operand of the circuit in addition to the size of its significand and its exponent according to the IEEE standard. The next three columns provide the runtimes. The timeout (TO in the table) is set to 24 hours. The experimental results clearly demonstrate the advantage of ACEC in verifying circuits that include data-path and control logic. While other equivalence checking tools can verify the correctness up to 16 bits, we are able to verify the correctness of a single precision binary floating-point multiplier (32 bit).

Table 8 shows some statistics about the algorithms of ACEC for checking the equivalence of the FP multiplier instances that contain the multiplier architecture (SP-WT-CH). For the reverse engineering algorithm, it shows the runtime of rewriting the combined model G; the runtime of extracting and abstracting data-path units; and number of the extracted units. These results show that the reverse engineering algorithm extracts more candidates for data-path units than the expected number. For two combined FP multipliers, six

1 It merges shareable resources into a single resource. A SAT solver is used to determine if two resources are shareable

Table 8: Statistics of ACEC for FP Multipliers

# bits	ACEC Algorithms		
	Reverse Engineering		
	Model Rewriting (h:m:s)	Extract & Abstract (h:m:s)	# Data-path Units
16	00:00:46	00:00:23	21
24	00:11:56	00:10:04	23
32	00:32:50	02:10:30	23
	Arithmetic Sweeping		
	# Variables of G'	# Proved Equivalences	Runtime (h:m:s)
16	1888	401	00:00:27
24	4440	666	00:03:36
32	5889	854	00:58:04
	Efficient Polynomial Representation		
	Decomposition		Logic Reduction
	# Reduced Terms	# Eff./Total Calls	# Canceled Terms
16	2400	514/3477	2916
24	9732	1013/8684	17153
32	16317	1345/12477	36390

data-path units should be extracted, two significand multipliers, two exponent adders, and two incrementers in the rounding stages. Also, the results show that reverse engineering spends more than 65% of the total time of ACEC.

For arithmetic sweeping, Table 8 gives total number of variables of the combined model; number of announced equivalences between variables of the two compared circuits; and the spent time by the sweeping algorithm. The results demonstrate that variables of G' which have functional similarities between each other account for less than 45% of the total number of variables.

Further, the table presents the number of saved terms by the decomposition of polynomials; number of effective (Eff.) calls for the decomposition algorithm wrt. the total calls for the algorithm (effective calls are those which save terms

of polynomials), and the number of canceled terms by the reduction rule which is applied during the model rewriting algorithm.

5.6 SUMMARY AND FUTURE WORK

In this chapter, we have presented a new algebraic equivalence checking technique for checking the equivalence of circuits that combine data-path and control logic. The technique utilizes a new reverse engineering algorithm to extract and abstract arithmetic components from the combined Gröbner basis representation of the compared circuits. Based on input and output boundaries of the abstracted components the proposed arithmetic sweeping deduces less and promising candidates for bit and word equivalences between the compared circuits. The technique circumvents the blow-up in number of terms of polynomials during the utilized algorithms by offering different types of decompositions for polynomials. Experimental results demonstrated the efficiency of our technique for the equivalence checking of large floating-point multipliers which cannot be verified with existing Boolean combinational equivalence checking techniques.

Directions for future work include:

1. Investigating the canonization of control logic, the ACEC can bring these units only to a semi-canonical form. Equivalence checking techniques based on BDDs and SAT are still more powerful to reason control logic circuits.

2. Managing the membership testing for non-equivalent circuits, whereas the number of calls for the IMT will increase, causing an exponential increase of the overall runtime of ACEC. Also, eliminating internal variables that have no equivalences may cause a blow-up in the size of the combined model.

3. Extending capabilities of the proposed reverse engineering approach, it cannot handle data-path circuits that their output words represent a redundant number system.

4. The assumption that MUXes appear only in the control logic is not always valid, more robust approaches to distinguish the control logic from the data-path logic are required.

6

CONCLUSIONS

This thesis resolves a hard problem that is beyond capabilities of state-of-the-art formal verification techniques, although it has been the subject of extensive investigations in the academia as well as the industry for almost two decades. The research in the thesis has aimed to propose a fully automated technique for formal verification of large scale bit-level arithmetic circuits, in particular, binary floating point. In fact, automatic formal verification for such circuits is unachievable using commercial as well as academic formal verification tools. To perform this task outstanding experts are employed, who exert an enormous amount of manual effort to understand the design and therefore decompose the verification problem into solvable cases. That why the goal of the thesis is significantly important to provide high quality systems that involve arithmetic circuits in a shorter time and a less cost.

To come up with such a full automated solution, the thesis has addressed first the hardest arithmetic unit to verify, which is the multiplier. Chapter 3 proposes the CPP approach that copes with the exponential complexity of verifying integer multipliers by decomposing automatically the verification problem into non-complex cases solved by standard equivalence checking tools. The approach does not only resolve the exponential complexity of the problem, but also it is more highly automated than the standard equivalence checking in the sense that it is not given any information about the high level description of the multiplier's design or even a golden reference that is compared against the multiplier.

Because the CPP approach is not applicable for all architectures of multipliers, the thesis investigates another formal verification technique based on concepts from the symbolic computation field. Chapter 4 introduces sophisticated algorithms that boost capabilities of the symbolic computation technique in order to verify large class of multiplier architectures including those that cannot be handled by the CPP approach. The enhanced technique is considered the most robust solution for the verification of large scale bit-level multipliers. Moreover, the major contribution of this chapter is that it draws the attention of the verification research community to consider decision procedures for formal verification purposes rather than classical solvers such as SAT/SMT.

The great success of the algebraic decision procedure provided by the enhanced symbolic computation—as shown in Chapter 4—was the key motivation to leverage this algebraic procedure in a full automated verification technique for binary floating-point circuits, in particular, floating-point multipliers. For this purpose, Chapter 5 offers the ACEC technique which is able of extracting from a given netlist of a floating-point circuit the necessary information that supports a decomposition procedure for the verification problem without any kind of personal intervention. To perform this task, a reverse engineering algorithm has been proposed to identify and abstract arithmetic components of the netlist. The abstracted information is utilized afterwards by the arithmetic sweeping algorithm to guide the decomposition of the overall problem. By proposing the ACEC technique, the thesis attains its challenging as well as long-term goal to verify large scale combinational arithmetic circuits in a fully automated manner.

Furthermore, the thesis can be extended in order to handle further problems that require solving nonlinear arithmetic constraints described at bit-level. It becomes obvious from the results of the thesis that utilizing bit solvers such as SAT and BDDs which express problems as propositional logic are not the panacea for all types of bit-level constraints, particularly, nonlinear arithmetic constraints. In contrast, algebraic solvers that manipulate polynomial representations of such constraints are more robust and more scalable. In a future work, an integration between the SAT solver and the algebraic solver should be considered in the sense that the propositional logic part of the problem is handled by the SAT solver, while the algebraic solver is leveraged for solving the arithmetic part. In the same time, the two solvers have to exchange their results in order to come up with a solution for the overall problem. Such a usage for the algebraic solver prompts its future enhancement by effective learning approaches such as the DPLL approach deployed by the SAT solver, where polynomials can be learned instead of clauses. Such improvements would expand the scope of algebraic solvers beyond digital circuits to further applications, e.g., verifying numerical programs as well as calculating their approximated errors [4, 33, 46], and analyzing cryptography algorithms [1, 7, 92].

A further future aspect is to modify sequential equivalence checking as well as model checking techniques to leverage the arithmetic information provided by a reverse engineering algorithm—similar to the one used by the ACEC technique—for verifying sequential circuits that their transition functions incorporate arithmetic components, e.g., multipliers. This development allows also a full automated verification for circuits such as floating-point division and floating-point square root.

BIBLIOGRAPHY

[1] J. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. "Certified computer-aided cryptography: Efficient provably secure machine code from high-level implementations." In: *ACM SIGSAC Conference on Computer & Communications Security*. 2013, pp. 1217–1230.

[2] *Arithmetic Module Generator Based on ACG*. available at `http://www.aoki.ecei.tohoku.ac.jp/arith/`. 2016.

[3] M. Aschenbrenner. "Ideal membership in polynomial rings over the integers." In: *Journal of the American Mathematical Society* 17.2 (2004), pp. 407–441.

[4] E. Barr, T. Vo, V. Le, and Z. Su. "Automatic detection of floating-point exceptions." In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Vol. 48. 1. 2013, pp. 549–560.

[5] J. Baumgartner, H. Mony, M. Case, J. Sawada, and K. Yorav. "Scalable conditional equivalence checking: An automated invariant-generation based approach." In: *Int'l Conf. on Formal Methods in CAD*. 2009, pp. 120–127.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. "Symbolic model checking without BDDs." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 1999, pp. 193–207.

[7] B. Blanchet. "Security protocol verification: Symbolic and computational models." In: *International Conference on Principles of Security and Trust*. 2012, pp. 3–29.

[8] K. Brace, R. Rudell, and R. Bryant. "Efficient implementation of a BDD package." In: *Design Automation Conf.* 1991, pp. 40–45.

[9] A. Bradley. "SAT based model checking without unrolling." In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. 2011, pp. 70–87.

[10] A. Bradley. "Understanding IC3." In: *International Conference on Theory and Applications of Satisfiability Testing*. 2012, pp. 1–14.

[11] D. Brand. "Verification of large synthesized designs." In: *International Conference on Computer-Aided Design*. 1993, pp. 534–537.

[12] R. K. Brayton. “The decomposition and factorization of Boolean expressions.” In: *IEEE International Symposium on Circuits and Systems*. 1982.

[13] R. Brayton and A. Mishchenko. “ABC: An academic industrial-strength verification tool.” In: *Computer Aided Verification*. 2010, pp. 24–40.

[14] M. Brickenstein and A. Dreyer. “PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials.” In: *Journal of Symbolic Computation* 44.9 (2009), pp. 1326–1345.

[15] R. E. Bryant. “Symbolic Boolean manipulation with ordered binary-decision diagrams.” In: *ACM Computing Surveys* 24.3 (1992), pp. 293–318.

[16] R. Bryant. “Graph-based algorithms for Boolean function manipulation.” In: *IEEE Transactions on Computers* 100.8 (1986), pp. 677–691.

[17] R. Bryant and Y. Chen. “Verification of arithmetic circuits with binary moment diagrams.” In: *Design Automation Conf.* 1995, pp. 535–541.

[18] R. Bryant and Y. Chen. “Verification of arithmetic circuits using binary moment diagrams.” In: *International Journal on Software Tools for Technology Transfer* 3.2 (2001), pp. 137–155.

[19] B. Buchberger. “Bruno Buchberger’s PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal.” In: *Journal of Symbolic Computation* 41.3 (2006), pp. 475–511.

[20] J. Burch, E. Clarke, K. McMillan, and D. Dill. “Sequential circuit verification using symbolic model checking.” In: *Design Automation Conf.* 1990, pp. 46–51.

[21] Y. T. Chang and K. T. Cheng. “Self-referential verification of gate-level implementations of arithmetic circuits.” In: *Design Automation Conf.* 2002, pp. 311–316.

[22] J. Chen and Y. Chen. “Equivalence checking of integer multipliers.” In: *ASP Design Automation Conf.* 2001, pp. 169–174.

[23] M. Ciesielski, C. Yu, D. Liu, and W. Brown. “Verification of gate-level arithmetic circuits by function extraction.” In: *Design Automation Conf.* 2015, 52:1–52:6.

[24] E. Clarke, M. Fujita, and X. Zhao. “Hybrid decision diagrams.” In: *International Conference on Computer-Aided Design*. 1995, pp. 159–163.

[25] J. Cortadella. “Timing-driven logic bi-decomposition.” In: *IEEE Transactions on Computer Aided Design of Circuits and Systems* 22.6 (2003), pp. 675–685.

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

[27] N. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge university press, 1980.

[28] M. Davis, G. Logemann, and D. Loveland. “A machine program for theorem-proving.” In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[29] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The COQ proof assistant: User’s guide: Version 5.6*. Tech. rep. Technical Report TR 134, INRIA, 1992.

[30] R. Drechsler, B. Becker, and S. Ruppertz. “The K* BMD: a verification data structure.” In: *IEEE Design & Test* 14.2 (1997), pp. 51–59.

[31] R. Drechsler, M. Herbstritt, and B. Becker. “Grouping heuristics for word-level decision diagrams.” In: *IEEE International Symposium on Circuits and Systems*. 1999, pp. 411–414.

[32] R. Drechsler and D. Sieling. “Binary decision diagrams in theory and practice.” In: *International Journal on Software Tools for Technology Transfer* 3.2 (2001), pp. 112–136.

[33] V. D’silva, D. Kroening, and G. Weissenbacher. “A survey of automated techniques for formal software verification.” In: *IEEE Transactions on Computer Aided Design of Circuits and Systems* 27.7 (2008), pp. 1165–1178.

[34] B. Dutertre and L. De Moura. “A fast linear-arithmetic solver for DPLL (T).” In: *Computer Aided Verification*. 2006, pp. 81–94.

[35] C. Eder and J. Perry. “F5C: a variant of Faugere’s F5 algorithm with reduced Gröbner bases.” In: *Journal of Symbolic Computation* 45.12 (2010), pp. 1442–1458.

[36] N. Een, A. Mishchenko, and R. Brayton. “Efficient implementation of property directed reachability.” In: *Int’l Conf. on Formal Methods in CAD*. 2011, pp. 125–134.

[37] N. Een and N. Sörensson. “An extensible SAT solver.” In: *Theory and Applications of Satisfiability Testing*. Vol. 2919. 2004, pp. 502–518.

[38] C. Van Eijk. “Sequential equivalence checking based on structural similarities.” In: *IEEE Transactions on Computer Aided Design of Circuits and Systems* 19.7 (2000), pp. 814–819.

[39] F. Farahmandi and B. Alizadeh. “Gröbner basis based formal verification of large arithmetic circuits using Gaussian elimination and cone-based polynomial extraction.” In: *Microprocessors and Microsystems* 39.2 (2015), pp. 83–96.

[40] J. Faugere. “A new efficient algorithm for computing Gröbner bases (F4).” In: *Journal of Pure and Applied Algebra* 139.1 (1999), pp. 61–88.

[41] M. Francis and A. Dukkipati. “Reduced Gröbner bases and Macaulay–Buchberger basis theorem over Noetherian rings.” In: *Journal of Symbolic Computation* 65 (2014), pp. 1–14.

[42] M. Fujita. “Verification of arithmetic circuits by comparing two similar circuits.” In: *Computer Aided Verification*. 1996, pp. 159–168.

[43] E. Goldberg, M. Prasad, and R. Brayton. “Using SAT for combinational equivalence checking.” In: *Design, Automation and Test in Europe*. 2001, pp. 114–121.

[44] G. Greuel, F. Seelisch, and O. Wienand. “The Gröbner basis of the ideal of vanishing polynomials.” In: *Journal of Symbolic Computation* 46.5 (2011), pp. 561–570.

[45] E. Guralnik, M. Aharoni, A. J. Birnbaum, and A. Koyfman. “Simulation-based verification of floating-point division.” In: *IEEE Transactions on Computer Aided Design of Circuits and Systems* 60.2 (2011), pp. 176–188.

[46] L. Haller, A. Griggio, M. Brain, and D. Kroening. “Deciding floating-point logic with systematic abstraction.” In: *Int’l Conf. on Formal Methods in CAD*. 2012, pp. 131–140.

[47] K. Hamaguchi, A. Morita, and S. Yajima. “Efficient construction of binary moment diagrams for verifying arithmetic circuits.” In: *International Conference on Computer-Aided Design*. 1995, pp. 78–82.

[48] J. Harrison. *The HOL Light Manual (1.1)*. Tech. rep. University of Cambridge Computer Laboratory, 1998.

[49] J. Harrison. “A machine-checked theory of floating point arithmetic.” In: *International Conference on Theorem Proving in Higher Order Logics*. 1999, pp. 113–130.

[50] "IEEE standard for floating-point arithmetic." In: *IEEE Std 754-2008* (2008), pp. 1–70.

[51] Bergeron J. *Writing Testbenches using SystemVerilog*. Springer Science & Business Media, 2007.

[52] C. Jacobi. "Formal verification of a fully IEEE compliant floating point unit." PhD thesis. Universität des Saarlandes, 2002.

[53] C. Jacobi and B. Christoph. "Formal verification of the VAMP floating point unit." In: *Formal Methods in System Design* 26.3 (2005), pp. 227–266.

[54] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner. "Automatic formal verification of fused-multiply-add FPUs." In: *Design, Automation and Test in Europe*. 2005, pp. 1298–1303.

[55] R. Kaivola and M. Aagaard. "Divider circuit verification with model checking and theorem proving." In: *International Conference on Theorem Proving in Higher Order Logics*. 2000, pp. 338–355.

[56] R. Kaivola and N. Narasimhan. "Formal verification of the Pentium® 4 floating-point multiplier." In: *Design, Automation and Test in Europe*. 2002, pp. 20–27.

[57] A. Kandri-Rody and D. Kapur. "Computing a Gröbner basis of a polynomial ideal over a Euclidean domain." In: *Journal of Symbolic Computation* 6.1 (1988), pp. 37–57.

[58] D. Kapur and Y. Cai. "An algorithm for computing a Gröbner basis of a polynomial ideal over a ring with zero divisors." In: *Mathematics in Computer Science* 2.4 (2009), pp. 601–634.

[59] M. Kaufmann, J. S. Moore, and R. S. Boyer. *A computational logic for applicative common lisp (ACL2)*. available at `http://www.cs.utexas.edu/users/moore/acl2/`. 2016.

[60] M. Keim, R. Drechsler, B. Becker, M. Martin, and P. Molitor. "Polynomial formal verification of multipliers." In: *Formal Methods in System Design* 22.1 (2003), pp. 39–58.

[61] A. KiranKumar, A. Gupta, and R. Ghughal. "Symbolic trajectory evaluation: The primary validation vehicle for next generation Intel® processor graphics fpu." In: *Int'l Conf. on Formal Methods in CAD*. 2012, pp. 149–156.

[62] I. Koren. *Computer Arithmetic Algorithms*. Universities Press.

[63] U. Krautz, V. Paruthi, A. Arunagiri, S. Kumar, S. Pujar, and T. Babinsky. "Automatic verification of floating point units." In: *Design Automation Conf.* 2014, pp. 1–6.

[64] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Science & Business Media, 2008.

[65] A. Kühlmann and F. Krohm. "Equivalence checking using cuts and heaps." In: *Design Automation Conf.* 1997, pp. 263–268.

[66] A. Kühlmann, V. Paruthi, F. Krohm, and M. K. Ganai. "Robust Boolean reasoning for equivalence checking and functional property verification." In: *IEEE Transactions on Computer Aided Design of Circuits and Systems* 21.12 (2002), pp. 1377–1394.

[67] D. Lundgren. *Double Precision Floating Point Core Verilog*. available at `http://opencores.org/project,double_fpu`. 2016.

[68] J. Lv, P. Kalla, and F. Enescu. "Efficient Gröbner basis reductions for formal verification of galois field multipliers." In: *Design, Automation and Test in Europe*. 2012, pp. 899–904.

[69] K. McMillan. "Interpolation and SAT-based model checking." In: *Computer Aided Verification*. 2003, pp. 1–13.

[70] P. Miner. "Defining the IEEE-854 floating-point standard in PVS." In: *Tech. Rep. TM-110167, NASA Langley Research Center* (1995).

[71] A. Mishchenko, S. Chatterjee, and R. Brayton. "DAG-aware AIG rewriting a fresh look at combinational logic synthesis." In: *Design Automation Conf.* 2006, pp. 532–535.

[72] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een. "Improvements to combinational equivalence checking." In: *International Conference on Computer-Aided Design*. 2006, pp. 836–843.

[73] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008, pp. 337–340.

[74] J. O'Leary, R. Kaivola, and T. Melham. "Relational STE and theorem proving for formal verification of industrial circuit designs." In: *Int'l Conf. on Formal Methods in CAD*. 2013, pp. 97–104.

[75] S. Owre, J. Rushby, and N. Shankar. "PVS: A prototype verification system." In: *International Conference on Automated Deduction*. 1992, pp. 748–752.

[76] J. O'Leary, X. Zhao, R. Gerth, and C. Seger. "Formally verifying IEEE compliance of floating-point hardware." In: *Intel Technology Journal* 3.1 (1999), pp. 1–14.

[77] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G. Greuel. "STABLE: A new QF-BV SMT solver for hard verification problems combining Boolean reasoning with computer algebra." In: *Design, Automation and Test in Europe*. 2011, pp. 1–6.

[78] T. Pruss, P. Kalla, and F. Enescu. "Efficient symbolic computation for word-level abstraction from combinational circuits for verification over finite fields." In: *IEEE Transactions on Computer Aided Design of Circuits and Systems* 35.7 (2016), pp. 1206–1218.

[79] E. Reeber and J. Sawada. "Combining ACL2 and an automated verification tool to verify a multiplier." In: *International Workshop on the ACL2 Theorem Prover and its Applications*. 2006, pp. 63–70.

[80] K. Rozier. "Linear temporal logic symbolic model checking." In: *Computer Science Review* 5.2 (2011), pp. 163–203.

[81] R. Rudell. "Dynamic variable ordering for ordered binary decision diagrams." In: *International Conference on Computer-Aided Design*. 1993, pp. 42–47.

[82] D. Russinoff, M. Kaufmann, E. Smith, and R. Sumners. "Formal verification of floating-point RTL at AMD using the ACL2 theorem prover." In: 2005.

[83] Y. Sato, S. Inoue, A. Suzuki, K. Nabeshima, and K. Sakai. "Boolean Gröbner bases." In: *Journal of Symbolic Computation* 46.5 (2011), pp. 622–632.

[84] A. Sayed-Ahmed, H. Fahmy, and U. Kühne. "Verification of the decimal floating-point square root operation." In: *European Test Symposium*. 2014, pp. 1–2.

[85] A. Sayed-Ahmed, U. Kühne, D. Große, and R. Drechsler. "Recurrence relations revisited: Scalable verification of bit level multiplier circuits." In: *IEEE Annual Symposium on VLSI*. 2015, pp. 1–6.

[86] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler. "Equivalence checking using Gröbner bases." In: *Int'l Conf. on Formal Methods in CAD*. 2016, pp. 169–176.

[87] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler. "Formal verification of integer multipliers by combining Gröbner basis with logic reduction." In: *Design, Automation and Test in Europe*. 2016, pp. 1048–1053.

[88] P. Seidel. "Formal verification of an iterative low-power x86 floating-point multiplier with redundant feedback." In: *International Workshop on the ACL2 Theorem Prover and its Applications*. 2011, pp. 70–83.

[89] H. Sharangpani and M. Barton. *Statistical analysis of floating point flaw in the pentium processor*. Tech. rep. Intel Corporation, 1994.

[90] A. Slobodová. "Challenges for formal verification in industrial setting." In: *International Workshop on Formal Methods for Industrial Critical Systems*. 2006, pp. 1–22.

[91] A. Slobodová, J. Davis, S. Swords, and W. Hunt. "A flexible formal verification framework for industrial scale validation." In: *Formal Methods and Models for Codesign (MEMOCODE)*. 2011, pp. 89–97.

[92] E. Smith and D. Dill. "Automatic formal verification of block cipher implementations." In: *Int'l Conf. on Formal Methods in CAD*. 2008, pp. 1–7.

[93] J. Smith and G. De Micheli. "Polynomial circuit models for component matching in high-level synthesis." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.6 (2001), pp. 783–800.

[94] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton. "Simulation graphs for reverse engineering." In: *Int'l Conf. on Formal Methods in CAD*. 2015, pp. 152–159.

[95] S. Stifter. "A generalization of reduction rings." In: *Journal of Symbolic Computation* 4.3 (1987), pp. 351–364.

[96] D. Stoffel and W. Kunz. "Equivalence checking of arithmetic circuits on the arithmetic bit level." In: *IEEE Transactions on Computer Aided Design of Circuits and Systems* 23.5 (2004), pp. 586–597.

[97] D. Stoffel, E. Karibaev, I. Kufareva, and W. Kunz. *Advanced Formal Verification*. Ed. by R. Drechsler. Kluwer Academic Publishers, 2004.

[98] G. Tseitin. "On the complexity of proofs in propositional logics." In: *Seminars in Mathematics*. Vol. 8. 1970, pp. 466–483.

[99] Y. Watanabe, N. Homma, T. Aoki, and T. Higuchi. "Application of symbolic computer algebra to arithmetic circuit verification." In: *Int'l Conf. on Comp. Design*. 2007, pp. 25–32.

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

[101] C. Wolf. *Yosys Open Synthesis Suite*. available at `http://www.clifford.at/yosys/`. 2016.

[102] B. Xue, P. Chatterjee, and S. K. Shukla. "Simplification of C-RTL equivalent checking for fused multiply add unit using intermediate models." In: *ASP Design Automation Conf.* 2013, pp. 723–728.

[103] C. Yu and M. Ciesielski. "Automatic word-level abstraction of datapath." In: *IEEE International Symposium on Circuits and Systems*. 2016.

[104] J. Yuan, C. Pixley, and A. Aziz. *Constraint-Based Verification*. Springer Science & Business Media, 2006.

[105] L. Zhang and S. Malik. "Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications." In: *Design, Automation and Test in Europe*. 2003, pp. 10880–10885.