FUNCTIONAL VERIFICATION OF ARITHMETIC CIRCUITS USING LINEAR ALGEBRA METHODS

A Thesis Presented

by

MOHAMED ABDUL BASITH AMEER ABDUL KADER

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

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2011

Electrical and Computer Engineering

FUNCTIONAL VERIFICATION OF ARITHMETIC CIRCUITS USING LINEAR ALGEBRA METHODS

A Thesis Presented

by

MOHAMED ABDUL BASITH AMEER ABDUL KADER

Approved as to style and content by:

Maciej Ciesielski, Chair

Israel Koren, Member

Eric Polizzi, Member

Christopher V. Hollot, Department Head
Electrical and Computer Engineering

To my family who moulded me into who I am today.

ACKNOWLEDGMENTS

I am grateful to the Almighty for his mercy and blessings. I would like to thank Professor Maciej Ciesielski who has been so inspiring as a guide and advisor and for providing unwavering and immeasurable support throughout the course of my research work. I would like to thank him for giving me this opportunity and for pushing me to perform to the best of my abilities. I am indebted to Dr.André Rossi, of Université de Bretagne-Sud, Lorient, France, who helped us develop the mathematical formulation and provided us with the necessary software, developed specifically for the project. I would also like to thank all my friends, room-mates and colleagues, especially my peers in the $VLSI - CAD$ Lab, for their unconditional help and support.

ABSTRACT

FUNCTIONAL VERIFICATION OF ARITHMETIC CIRCUITS USING LINEAR ALGEBRA METHODS

SEPTEMBER 2011

MOHAMED ABDUL BASITH AMEER ABDUL KADER

B.E., ANNA UNIVERSITY, INDIA

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Maciej Ciesielski

This thesis describes an efficient method for speeding up functional verification of arithmetic circuits namely linear network such as wallace trees, counters using linear algebra techniques. The circuit is represented as a network of half adders, full adders and inverters, and modeled as a system of linear equations. The proof of functional correctness of the design is obtained by computing its algebraic signature using standard linear programming (LP) solver and comparing it with the reference signature provided by the designer. Initial experimental results and comparison with Satisfiability Modulo Theorem (SMT) solvers show that the method is efficient, scalable and applicable to complex arithmetic designs, including large multipliers. It is intended to provide a new front end theory/engine to enhance SMT solvers.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Formal Verification

With the increasing size and complexity of integrated circuits (IC) and systems on chip (SoC), design verification has quickly become a dominating factor of the overall design flow. In today's designs over 75% of the entire design effort and cost is devoted to verification, making it the most challenging and expensive part of the overall design process. Of particular importance (and difficulty) is verification of arithmetic datapaths and their components, such as multipliers. Checking for functional correctness of arithmetic circuits and datapaths remains a major concern in IC and SoC design flows.

In early stages of CAD for synthesis and verification, simulation was a predominant technique for design verification. Simulation is based on computing the output values for a sequence of input patterns, and as such is an exhaustive technique. Its success depends on computing response to all input patterns and hence there is every possibility that there might be design errors which may never get identified during logic simulation. Also, as the size and complexity of the design grows, the number of input pattern grows exponentially which makes it impossible to verify the overall design using logic simulation. These drawbacks have led to the develpment of Formal Verification techniques which are based on proving global mathematical properties of the design rather than simulating circuit responses for a particular input sequence.

The most promising approach to verification of arithmetic designs is formal verification, which proves functional correctness of a design using mathematical logic and

formal reasoning instead of simulation. Various techniques for functional verification, such as Binary Decision Diagrams (BDDs), Boolean satisfiability checking (SAT) methods, model checking, property checking, and equivalence verification, have been used as a proof process. An overview of a formal verification flow is shown in Figure 1.1

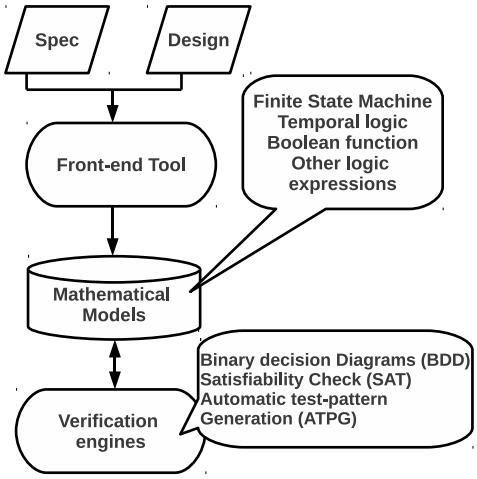


Figure 1.1. Formal verification flow.

As shown in Figure 1.1, in formal verification environment the specification and design description are first converted into mathematical models. These models include finite state machines, temporal logic, Boolean functions and other logic expressions. The mathematical models are then verified using different canonical representations, satisfiability checks and automatic test-pattern generation techniques. Unlike logic simulation, which requires exhaustive check of design responses to all possible input combinations, formal methods use mathematical models of the design to prove its properties and functionality.

1.1.1 Equivalence Checking

Equivalence Checking or Equivalence Verification is concerned with verifying whether two designs are equivalent or not. Equivalence checking can be applied to two designs at the same design level or at different design levels. Based on the type of equivalence definition, equivalence checking can be applied to either combinational parts of the design or to the sequential part. Today, combinational designs as large as having 10 million gates can be efficiently verified by contemporary equivalence verification tools with the notable exception for complex arithmetic circuits.

A number of canonical graph-based representations are used to solve equivalence checking for combinational designs. They include decision diagrams such as Binary Decision Diagrams (BDDs) [6] and their many variants, Binary Moment Diagrams (BMDs) [7], Taylor Expansion Diagrams (TEDs) [11] and others. Since BDDs and BMDs cannot efficiently represent abstract designs, Taylor Expansion Diagrams (TEDs) and Finite-field Decision Diagrams (FFDDs) were introduced. These are extensions of BDDs and BMDs with input and output represented as symbolic variables. The difference between the two classes of diagrams lies in the arithmetic representation of data; TEDs use integer arithmetic, whereas FFDDs use finite field arithmetic. They also differ in the type of decomposition used; TEDs use Taylor series expansion whereas FFDDs use multi-valued Galois field (GF) decomposition. These canonical representations are reviewed in the next chapter.

1.1.2 Boolean Satisfiablity

Boolean Satifiability (SAT) [20] is a constraint satisfaction problem that appears in computer-aided design of VLSI circuits and in Artificial Intelligence. Given function ϕ, SAT checks for a variable assignment for which function ϕ is true. If there exists such an assignment of variables then ϕ is satisfiable, else it is unsatisfiable. Despite the fact that Boolean Satisfiability is NP-complete, SAT is used in many

real world problems including test pattern generation, equivalence checking, testing, logic synthesis, logic simulation and others. Most SAT solvers use Conjunctive Normal Forms (CNFs). The goal is to prove that there are no input patterns for which the outputs of the two designs differ. This can be solved by connecting the outputs through a "miter", F, that includes an XOR gate as shown in Figure. 1.2. If the output of a miter $F = 1$ is satisfiable, the two designs are different, and the SAT solver provides an example of input patterns(counter-proof) for which $F_1 \neq F_2$. Otherwise, such constructed SAT problem is Unsatisfiable (unSAT), in which case the designs are equivalent. In summary, the equivalence problem is reduced to proving Unsatisfiability of such a structure.

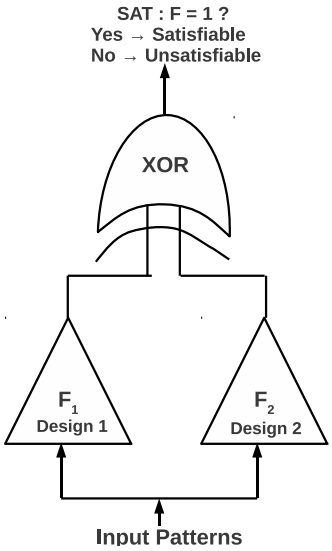


Figure 1.2. An example of SAT.

1.1.3 Model Checking

Model Checking [8], or Property Checking, is a technique which determines whether the design satisfies the properties given in its specification. A model checking tool accepts system requirement (model) and a property (specification) to ensure that the model and the property are satisfied. The model checking tool returns "true" if

the model satisfies the property, otherwise it returns a counter-example. The idea of providing a counter-example is to detail the reason for the property not being satisfied by the model. Thus by analyzing the counter-example the source of error can be found and rectified and the model can be tried again. This repetitive process of checking if a given property is satisfied and generating counter-example ensures that several properties are satisfied giving confidence in the correctness of the design. Over the years, model checking has undergone remarkable improvement owing to the improvement in SAT techniques [20]. The basic idea of model checking approach is illustrated in Figure 1.3

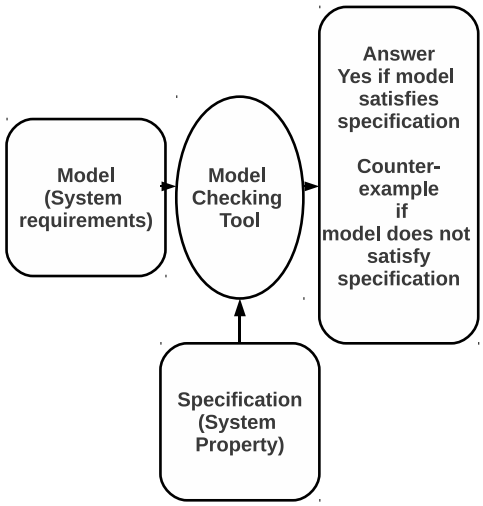


Figure 1.3. Model Checking.

1.1.4 Automatic Test Pattern Generation

Automatic Test Pattern Generation (ATPG) [10] is a technique of generating test vectors as stimulus for testing the circuits for manufacturing faults. Manufacturing faults are physical defects in the circuit generated during manufacturing process, which result in incorrect operation of the circuit. The faults led to the circuit sig-

nals being permanently stuck at either logical 0 or 1. Several efficient algorithms have been developed for stuck-at faults detection using Boolean reasoning. Thus using Automatic Test Pattern Generation techniques, powerful formal verification techniques can be formed. Automatic test pattern generation is often used in conjunction with SAT.

1.2 Motivation

Unlike gate-level logic designs, which can be handled using Boolean methods, arithmetic designs require treatment on higher abstraction levels. Techniques based on decision diagrams or SAT solvers that work at the circuit and logic level are not scalable for complex arithmetic systems as they require "bit-blasting", i.e., flattening of the entire design into bit-level netlists. Some of the recent approaches to verification use Satisfiability Modulo Theories (SMT) and symbolic algebra methods, but they suffer from lack of adequate models that can harness the inherent bit-level nature of complex arithmetic designs.

The work described in this thesis aims at overcoming some of these limitations. It presents a novel approach to perform functional verification of arithmetic circuits, which combines algebraic description of the arithmetic design with the bit-level details, using linear algebra techniques. The proof of correctness is obtained by modeling the arithmetic circuit as a network of half adders and full adders and computing an algebraic signature of the circuit using a standard LP solver. The computed signature is then compared with the reference signature (golden model) provided by the designer to determine if the design is correct. The computation of the algebraic signature is very fast and scalable as it is based on a simple manipulation of a set of linear equations without finding variable assignments (typical of SAT approaches) and without imposing integer constraints on its variables. In addition to functional verification and property checking, the method can be used to extract circuit behav-

ior from its structural description by computing its input signature. The proposed technique can be used in conjunction with, or as one of the SMT engines (theories) to enhance capabilities of the current SMT solvers.

CHAPTER 2

PREVIOUS WORK

Several methods have been proposed to check an arithmetic circuit against its specification at a higher level of abstraction. Different variants of canonical graph-based representations have been proposed, including Binary Decision Diagrams (BDDs) [6] or their variants such as Multiplicative Power Hybrid Decision Diagrams (*PHDDs) [9], Hybrid Decision Diagrams (HDDs) [12], etc. These representations are commonly referred to as Decision Diagrams. Decision diagram is a graph based structure where the nodes of the graph represent the variables and the edges represent the decomposition of the function with respect to the individual variables.

An important feature of decision diagrams is canonicity, meaning that the representation of a function with such a diagram is unique. Two combinational circuits can be checked for equivalency by simply checking if their decision diagrams are isomorphic. In practice, the decision diagrams for the two functions are built in the same manager, so the test for isomorphism reduces to checking if the two functions point to the same root of the diagram.

More advanced methods include Binary Moment Diagrams (BMDs) [7] and Taylor Expansion Diagrams (TEDs) [11] that attempted to represent the design at higher levels of abstraction. The next section reviews the different canonical diagram representations.

2.1 Binary Decision Diagrams (BDDs)

The most commonly used decision diagram representation is Binary Decision Diagram (BDD). [6] BDD is a canonical representation of Boolean functions represented as a directed acyclic graph. It provides a more compact representation than truth-tables, Sum of Products (SOPs) form, or Conjunctive Normal Forms (CNFs) for most of the Boolean functions used in the area of VLSI design. BDDs are based on the Shannon function decomposition, where the function is decomposed into two co-factors $f_x = 0$ and $f_{\overline{x}} = 1$. Individual paths lead to taking a decision *x=0* or *x=1*, hence the name "decision diagrams". Efficient algorithms exist to represent Boolean functions as BDDs. BDDs represent Boolean functions and logic circuits at bit-level and are used to verify bit-level designs, such as control and random logic. They are extensively used in verification and logic synthesis but also in satisfiability and testing. Bryant [6] proposed an algorithm to reduce ordered BDDs, known as Reduced Ordered BDDs (ROBDDs). This form is irredundant, canonical and minimal.

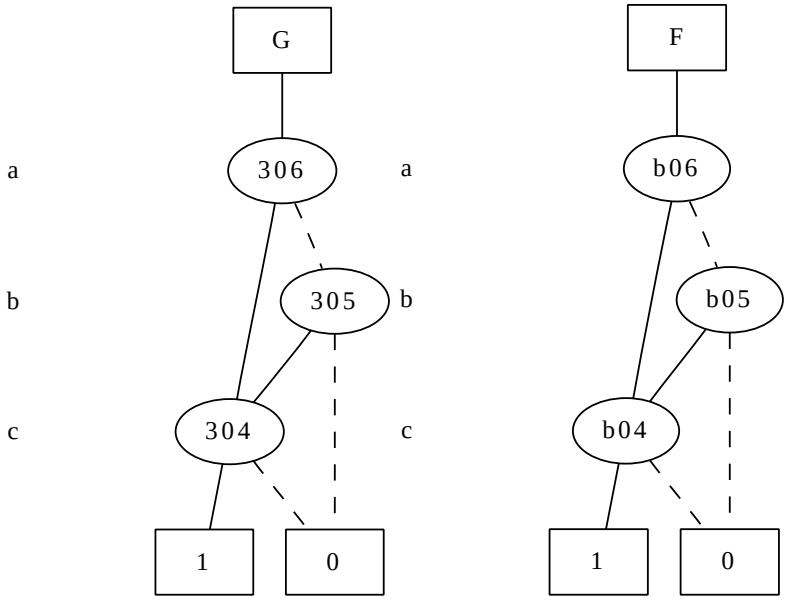


Figure 2.1. Two equivalent expressions having the same BDD. (a)$F = a'bc + abc + ab'c$; (b) $G = ac + bc$

An example of use of BDD in equivalence checking is shown in Figure 2.1. The two function F and G are equivalent and hence their BDDs are identical (for the same ordering of variables).

Another application of BDDs is in solving Boolean Satisfiability (SAT), illustrated in Figure 2.2 with a BDD for $F = a(b + c)$. In-order to find a satisfiability solution for $F = 1(0)$ one needs to find a path from the root of the BDD to terminal node 1 (0). In this case the paths ab or $ab'c$ provide a satisfiability solution, i.e., $a = b = 1$ or $b = 0$, $a = c = 1$ satisfies $F = 1$.

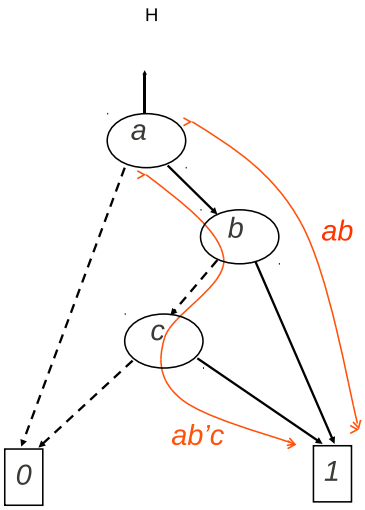


Figure 2.2. Application of BDD to SAT.

BDDs have emerged as data structure of choice in logic synthesis, SAT, and Boolean logic representation, but their application to verification of arithmetic circuits is limited. In general, solutions based on bit-level decision diagrams suffer from high computational complexity due to exponential growth of the BDD size for complex arithmetic circuits such as multipliers. For example, a BDD for 14×14 multiplier

cannot be built in a typical modern computer due to large memory size required by its data structure. Recognizing this weakness, researchers turned to other canonical forms, such as BMDs, TEDs, and others capable of representing designs at higher levels of abstraction than a bit-level.

2.2 Binary Moment Diagrams (BMDs)

Another form of canonical graph-based diagrams that are more applicable to arithmetic functions than BDDs are Binary Moment Diagrams (BMDs) introduced by Bryant [7]. BMDs are based on moment decomposition principle that treat arithmetic functions as linear functions with Boolean input and integer output. BMDs are used in verifying arithmetic designs with bit-level inputs and integer outputs.

BMDs use modified Shannon's expansion, where the Boolean variable is treated as a binary(0,1) integer variable. The complement of x is modeled as $\overline{x} = 1 - x$ and the terms of the expansion are regrouped as

$$f(x) = (1 - x) \cdot f_{\overline{x}} \; + \; x \cdot f_x = f_{\overline{x}} \; + \; x \cdot (f_x - f_{\overline{x}}) = f_{\overline{x}} \; + \; x \cdot f_{\Delta x}$$

Here "·", "+" and "-" denote multiplication, addition and subtraction, respectively. The above decomposition is termed as *moment* decomposition where $f_{\overline{x}}$ is the *constant moment*, and ($f_{\Delta x} = f_x - f_{\overline{x}}$) is the *linear moment*. Thus, a Boolean function f is treated as a *linear function* in x, with $f_{\overline{x}}$ as the constant term, and $f_{\Delta x}$ as the linear coefficient of f, the partial derivative of f with respect to x. This transformation relies on the fact that the variable x is still Boolean, i.e., it evaluates to either 0 or 1.

Each node of a BMD represents a function based on its moment decomposition, as shown in Figure 2.3(a). The two edges coming from a node represent the constant moment (shown in dashed lines) and the first moment (shown in solid lines) of the

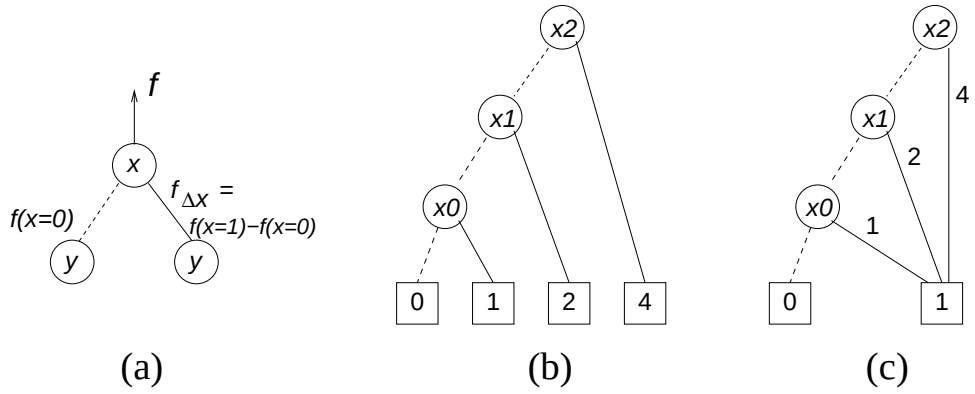


Figure 2.3. Binary Moment Diagrams: (a) The moment decomposition principle; (b) BMD for binary encoded integer $X = 4x_2+2x_1+x_0$; (c) *BMD for $X = [x_2\ x_1\ x_0]$.

function with respect to the decomposing variable. The BMD representation of the unsigned integer $X = 4x_2 + 2x_1 + x_0$ encoded with $n = 3$ bits is shown in Figure 2.3(b). The three constants at the terminal nodes of the BMD can be moved to the edges and represented as edge-weights, as shown in Figure 2.3(c). This diagram is known as *multiplicative binary moment diagram* or *BMD. There are two major features that differentiate *BMDs from decision diagrams.

- BMDs are not decision diagrams since they are based on the moment decompositions and not on the bit-wise Shannon expansion.
- BMDs are multiplicative diagrams, i.e., each path from the root to the terminal node is a product of the variables labeling the nodes and edge weights along the path. As such, they are not applicable to SAT.

In addition to integer word-level functions, a *BMD can be used to represent Boolean logic. The equations used to model Boolean logic are as follows:

$$NOT: \quad x' = \quad (1 - x) \tag{2.1}$$

$$AND: \quad x \wedge y = \quad x \cdot y \tag{2.2}$$

$$OR: \quad x \vee y = \quad x + y - x \cdot y \tag{2.3}$$

$$XOR: \quad x \oplus y = \quad x + y - 2x \cdot y \tag{2.4}$$

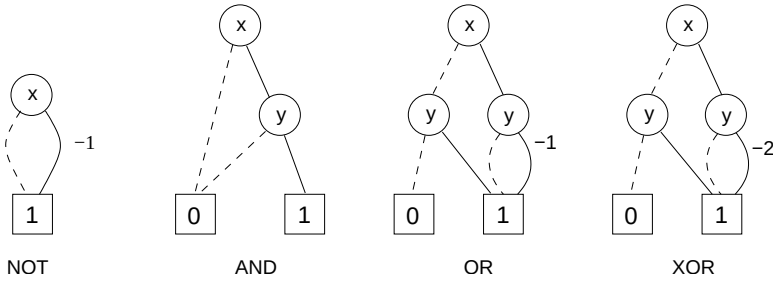


Figure 2.4. *BMD representation for Boolean operators: a) NOT: $x' = (1 - x)$; b) AND: $x \wedge y = x \cdot y$; c) OR: $x \vee y = x + y - xy$; d) XOR: $x \oplus y = x + y - 2xy$.

Figure 2.4 shows BMD representations for these basic Boolean operators [7]. In the diagrams, x and y are Boolean variables represented by binary variables, and $+$ and $\cdot$ represent algebraic operators of ADD and MULT, respectively. The resulting functions are 0,1 integer functions.

In principle, a word-level BMD can efficiently represent integer multiplication, since its size grows linearly with the size of the multiplier's input. However, BMDs require word-level information about the design, which is usually not available or is hard to extract from optimized bit-level implementation. Also, an integer (word-level) output of a BMD cannot be split into individual bits, which severely limits its application to arithmetic, bit-level verification. Another limitation is that BMDs cannot be used for solving SAT problems since they are multiplicative diagrams, i.e., the weights combine multiplicatively along the path from the terminal node to the root. Solving integer-valued SAT problem in this structure is similar to solving integer factorization problem which is known to be hard.

2.3 Taylor Expansion Diagrams (TEDs)

To address some of the limitations of BMDs and BDDs, and in particular the need for a more abstract representation of designs with arithmetic components, another type of diagram, Taylor Expansion Diagram (TED), has been introduced [11]. TEDs are based on Taylor series expansion of polynomial representation of the computations

expressed by the design. TED representation maps word-level inputs into word-level outputs and represents the infinite precision computation as polynomial. TEDs can be used in verification of designs at behavioural and algorithmic levels, such as datapath and signal processing systems, due to their power of abstraction, canonicity and compactness. The computations in such designs are expressed as polynomials and represented with TEDs, with memory requirements orders of magnitude smaller than those of other representations. TEDs can also be used to transform the initial functional representation into a structural representation, such as data-flow graphs (DFG), which makes them applicable to behavioural and high-level synthesis.

Construction of a TED for an RTL design starts with building trivial TEDs for its primary inputs. Partial expansion of the word-level input signals is often necessary when one or more bits from any of the input signals fan out to other parts of the design. This is the case in the designs shown in Fig. 2.5 (a) and (b), where bits $a_k = A[k]$ and $b_k = B[k]$ are derived from word-level variables A and B. In this case, the word-level variables must be decomposed into several word-level variables with shorter bit-widths.

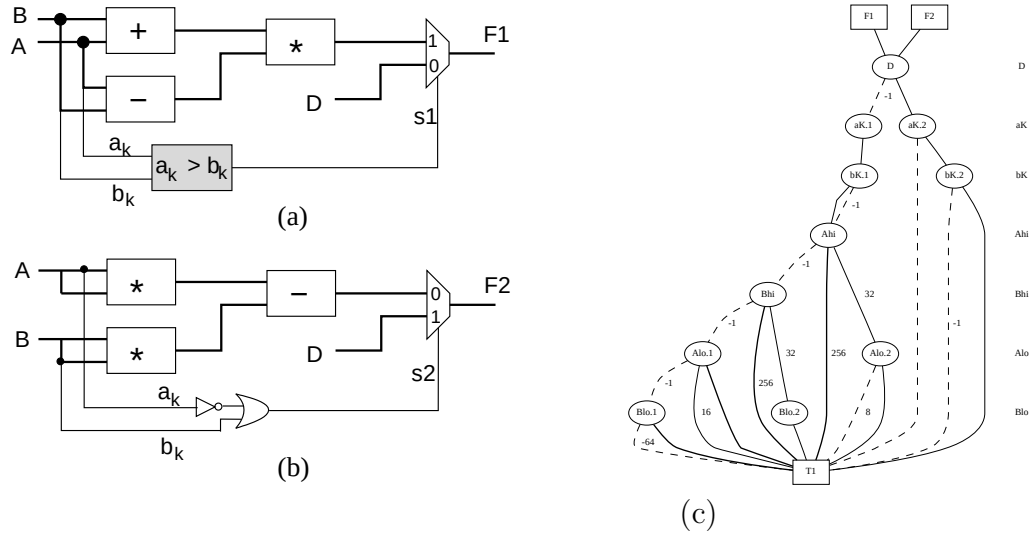


Figure 2.5. RTL verification using canonical TED representation: (a), (b) Functionally equivalent RTL modules; (c) The isomorphic TED for the two designs.

Once all the abstracted primary inputs are represented by their TEDs, Taylor Expansion Diagrams can be constructed for all the components of the design. TEDs for the primary outputs are then generated by systematically composing the constituent TEDs in the topological order, from the primary inputs to the primary outputs. For example, to compute $A + B$ in Fig. 2.5 (a) and (b), the ADD operator is applied to functions A and B (each represented in terms of their abstracted components). The subtract operation, $A - B$, is computed by first multiplying B with a constant -1 and adding the result to the TED of A. The multipliers are constructed from their respective inputs using the MULT operator, and so on. To generate a TED for the output of the multiplexors, the Boolean functions s_1 and s_2 first need to be constructed as TEDs. Finally, the TEDs for the primary outputs are generated using the MUX operator with the respective inputs. As a result of such a series of composition operations, the outputs of the TED represent multi-variate polynomials in terms of the primary inputs of the design.

After having constructed the respective ordered, reduced, and normalized Taylor Expansion Diagram for each design, the test for functional equivalence is performed by checking for isomorphism of the resulting graphs. In fact, the TED-based verification is similar to that using BDDs and BMDs: the generation of the TEDs for the two designs under verification takes place in the same TED manager; when the two functions are equivalent, both functions point to the same root of the common TED, shown in Fig. 2.5(c).

It should be noted that the arithmetic operations in these designs assume that no overflow is produced by any of the intermediate signal. That is, functions $F1$ and $F2$ are functionally equivalent under an *infinite precision* computation model. This limitation is a natural consequence of the design representation on the abstract level, where notion of the individual bits is not available. The major limitation is

that TEDs cannot express individual output bits (eg; sign) as function of word-level inputs. Splitting of signals is a problem as well.

2.4 Symbolic Algebra

Computations encountered in behavioural design specifications such as digital signal and image processing designs, digital filter designs, and designs that employ complex transformations, such as DCT, DFT, WHT, etc. can be represented in terms of polynomials. For the purpose of component mapping, polynomial models of high level design specifications have been used in behavioural synthesis in [32, 33, 26]. For each component (operator), a polynomial representation is created and the polynomials are matched by comparing their co-efficients. However, for large multivariate polynomials, comparing and storing large matrices becomes inefficient.

Commercial symbolic algebra tools, such as Maple [40], Mathematica [41], and MatLab [43], use advanced symbolic algebra methods to perform efficient manipulation of mathematical expressions, including fast multiplication, factorization, etc. These tools have also been used for the purpose of polynomial mapping, namely, to perform simplification modulo polynomial as in [26]. However, despite the unquestionable effectiveness of these methods for classical mathematical applications, they are less effective in modeling of large scale digital circuits and systems, and in particular in polynomial verification. For example, symbolic algebra tools offered by Mathematica and alike cannot unequivocally determine the *equivalence* of two polynomials. The equivalence is checked by subjecting each polynomial to a series of *expand* operations and comparing the coefficients of the two polynomials ordered lexicographically.

Recently, a number of computer symbolic algebra methods have been applied to model RTL designs with arithmetic components as polynomial expressions, mostly for equivalence checking. A polynomial abstraction technique based on the fundamental

theorem of algebra has been proposed in [27]. This abstract model preserves the control and data properties of the original system and can be verified via symbolic model checking. Some of the newer approaches combine symbolic computation methods with the relational modeling techniques from Kleene algebra [17],[50]. In [51] both the data path and the control part of the design are encoded using polynomials, and the verification is performed in a generalized bounded model checking style.

A concept of polynomial functions over finite integer rings has been introduced in [30] to perform equivalence verification of arithmetic data-paths with fixed bit-width. Similar to BMDs, its applicability to verification of arithmetic circuits is limited as it relies on a word-level representation of the datapaths, which is often not available. An attempt to address modular datapath verification and optimization problem is discussed in [1]. It uses Horner expansion diagram (HED) to reduce and prove equivalence of portions of datapaths modeled as polynomials over finite integer rings. This work is limited to relatively small DSP designs, such as filters, that can be represented as polynomials of higher degree. It basically suffers from the same problem as TED. An approach to verification of bit-level implementations using theory of Grobner basis over fields has been proposed by [47] and adopted by others [31, 49]. These methods are very complex and limited to small polynomials.

Kalla [31] proposed a method of equivalence checking of two polynomials implemented in n-bit representation by checking if the difference between the two expressions is a "vanishing polynomial", i.e., if it reduces to 0 modulo n. This method, however, relies on a complicated theory of Grobner basis and requires the use of symbolic algebra systems such as Maple. Furthermore, the "vanishing polynomial" method cannot be easily used to compare two complex arithmetic designs expressed on arithmetic Boolean level and with logic gates.

A technique based on term rewriting was proposed [46] for RTL equivalence checking, using a database of rewrite rules for typical multiplier implementation schemes.

However, the method cannot be automated for non-standard implementations. Automated techniques for extracting arithmetic bit level information from gate level netlists has been proposed in the context of equivalence checking [45] and debugging [29].

2.5 Arithmetic Bit-Level (ABL)

Automated techniques for extracting arithmetic bit level information from gate level netlists has been proposed in the context of equivalence checking [45] and debugging [29]. In [45, 49] a gate level network of an addition circuit (a basic component of the multiplier) is modeled as a network of half adders, called *arithmetic bit-level* (ABL) network, and a heuristic ABL normalization was added to enable their comparison [48]. Since ABL representation is not unique, a heuristic *ABL normalization* was proposed to enable their comparison [48]. Basically, the ABL description of addition networks and bit-wise multiplication is transformed into a reduced normal form, using commutative and distributive laws. Such normalization process enables structural similarities between the design under verification and the specification given as a property, which simplifies job for standard equivalence checkers. In [49] ABL normalization is combined with the techniques of computer algebra to produce a set of equivalent "variety subset problems". ABL components are modeled by polynomials over unique ring, and the normal forms are computed w.r.t. the Grobner basis over rings $Z/2^n$ using modern computer algebra algorithms. ABL descriptions can be also derived from the gate-level implementations using Red-Muller decomposition techniques, but they yield unduly large polynomials of high degree. All these methods are computationally intensive and are not scalable to large designs.

A simplified version of this technique has been recently presented by the same group [25], replacing expensive Grobner base computation by direct generation of polynomials representing individual output bits in terms of the primary inputs. How-

ever, no general method for deriving such (potentially very large) polynomials and comparing them in a systematic way against the specification has been proposed.

A high-level multiplier description language has been introduced in [18] to model a wide range of common implementations at a structural (gate-level) and arithmetic level. The correctness of the created model is established by bit-level transformations matching the model against a standard multiplication specification. The model is also translated into a gate level netlist to be compared with the implementation using standard equivalence checker. The method relies on a large amount of structural similarity between the two models that enables the use of standard equivalence checkers.

It is important to emphasize that this approach relies on the existence of structural similarities, which is common in standard equivalence checkers in industry. However, a pair of designs to be checked for equivalence are often structurally different. For example, word-level signals at the output of first level boxes (Add, Sub in Figure 2.5 (a) and *,* in (b)) look structurally similar, but are not equivalent ($A+B, A-B$ vs A^2, B^2). This makes a verification tool that relies on such similarity difficult. This can be illustrated with a pair of designs in Figure 2.5. In contrast, no assumption is made in our work about structural similarity; instead, the formal proof for functional equivalence is provided using mathematical model and specifically, standard linear algebra.

2.6 Satisfiability Modulo Theories (SMT)

Another direction of research into design verification investigates Satisfiability Modulo Theory (SMT) solvers. Satisfiability Modulo Theories (SMTs) solve formal properties at higher level than Boolean functions. SMT is a generalization of Boolean SAT, in which the binary variables are replaced by predicates or binary-valued functions of non-binary variables. These predicates may come from a variety of under-

lying theories. SMT solvers combine Boolean SAT with specialized solvers for some well-defined theories, such as Boolean logic, linear arithmetic, theory of equality of uninterrupted functions, theory of bit-vectors, theory of arrays and list structures and others [4] [19]. SMTs can enrich Conjunctive Normal Form (CNF) with linear constraints, arrays, all-difference constraints, uninterpreted functions, etc. SMT solvers use the same engine as SAT, but integrates theory reasoning and Boolean reasoning. Despite great advances of SAT solvers in verification, notably in property checking, their applicability to functional verification of highly-optimized custom datapath implementation remains limited [49].

Some of the SMT solvers for quantified free linear arithmetic include MathSAT [37], YICES [38], Z3 [39], and for non-linear arithmetic ABSolver [3] and iSAT [36]. New SMT solvers are developed continually, including those to efficiently solve linear inequalities in integer domain (Mistral) [16]. Our work is related to this topic but tackles the problem in a way that does not require solving a decision problem or finding a satisfying assignment for the integer variables.

2.7 Summary

In summary:

• Standard verification techniques based on decision diagrams and Boolean SAT are unable to handle arithmetic functions efficiently, especially multipliers.

• ILP methods are ineffective, as they inevitably require finding satisfying assignments for all *integer* or *binary* variables. Such a process is known to be computationally prohibitive for large designs.

• Current symbolic algebra methods are limited to relatively small designs that can be represented as polynomials. Advanced methods based on Grobner theory are very complex, and efficient algorithms for solving the associated verification problem are not readily available.

• Standard industry practice that performs functional verification by bringing the implementation and specification to a common level (typically gate-level) and using equivalence checking, relies on structural similarity between the two designs, which is often absent. The same is true for current ABL methods reviewed here.

In contrast, **in this thesis** no assumption is made about internal structural similarity between the implementation (arithmetic circuit) and the specification (its intended function). In fact, the specification is given just as a simple linear expression that relates word-level output to bit-level inputs. Such a specification can be trivially derived for all the known arithmetic designs, such as different adders, simple and recoded multipliers, etc. The only requirement is that the design is arithmetic, combinational, and the designer knows its intended function.

In this work, the functional verification is solved using standard *linear algebra* techniques. In contrast to the existing ILP methods it does not require solving an ILP problem. Instead, the computation is based on a simple manipulation of a set of linear equations, similar to Gaussian elimination, without performing variable assignments (typical of SAT approach) and without imposing integer constraints on its variables. This method can be used to enhance capabilities of current SMT solvers by integrating it as one of the SMT engines (theories).

Thus it can be seen that verification of arithmetic designs remains an unsolved problem and that it could be solved only at the level that incorporates Boolean (bit) level view of the design.

CHAPTER 3

ALGEBRAIC MODELING OF ARITHMETIC NETWORKS

3.1 Arithmetic Network

It has been shown that any arithmetic circuit can be expressed as a network of half-adders (HA), full-adders (FA) and basic logic gates (XOR, AND, OR, and INV) [45]. For this reason, in this work we will refer to such a representation as an *arithmetic network*, and we will use the terms *circuit* and *network* interchangeably. Furthermore, logic gates can also be converted to a simple combination of HA or FA circuits [49]. As a result, any arithmetic circuit can be expressed solely in terms of HA, FA and inverters. For the purpose of this work, an arithmetic network is represented as a directed graph $G = (V, E)$, where V is a set of vertices and E is a set of directed edges. Vertex $v \in V$ models an arithmetic operator (HA, FA) or inverter. An edge $e \in E$ represents an electrical signal connecting two vertices. Each signal in the network is represented by a variable x_k.

Each arithmetic or logic operator is modeled with a set of linear equations that involves variables representing its input and output signals. For example, a half-adder (HA) with binary inputs $a,\ b$, binary outputs S (sum) and C (carry out), is modeled as

$$a + b = 2C + S \tag{3.1}$$

Similarly, a full adder (FA) with binary inputs $a,\ b,\ c_{in}$, binary outputs S and C is modeled as

$$a + b + c_{in} = 2C + S \tag{3.2}$$

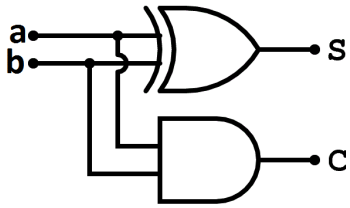


Figure 3.1. Logic-level Half-adder, HA(a, b)

Logic gates can be similarly represented by algebraic equations by deriving their functions from a half adder. This can be seen from Figure 3.1 of a logic-level half adder circuit. Specifically, XOR(a, b) is simply a sum output, S, of the half adder HA(a, b), and the AND(a, b) is the carry-out output, C, of HA(a, b).

The OR gate, $d =$OR(a, b), can be similarly derived (using deMorgan's law) from the carry out (AND) output of the HA by inverting its inputs and outputs:

$$(1 - a) + (1 - b) = 2(1 - d) + S \tag{3.3}$$

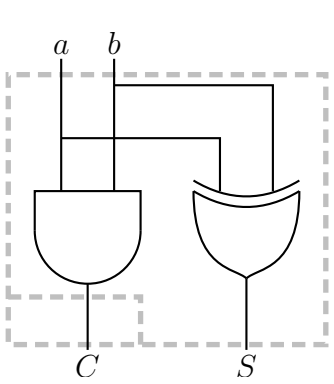
Or, equivalently

$$a + b = 2d - S \tag{3.4}$$

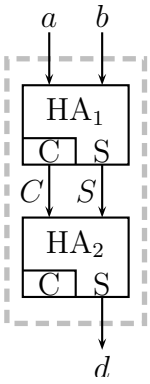
Combining this equation with the equation (3.1) for HA gives $C + S = d$. As a result, an OR(a, b) gate can be modeled with the following equations involving two half adders:

$$\begin{cases} a + b = 2C + S \\ C + S = d \end{cases} \tag{3.5}$$

Figure 3.2 shows HA model for the basic logic gates, AND, OR and XOR. The correctness of the equations can be verified with the truth table provided in the figure.



(a)



(b)

a	b	C	S	d
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

(c)

Figure 3.2. Modeling of logic gates using half-adders: (a) S =XOR(a, b), C =AND(a, b) of half-adder; (b) OR gate $d = C + S$ derived from two half-adders; (c) truth table for $C,\ S,\ d$.

Finally, the inverter gate y =INV(x) can be trivially modeled by the following equation: $x + y = 1$.

3.2 Mathematical Model of Arithmetic Network

The functionality of an arithmetic network composed of HA, FA and INV nodes can be represented by a system of linear equations whose variables are inputs x_I, outputs x_O and internal signals x_S. There is one equation for each HA, FA, XOR gate or AND gate (3.1) and (3.2), and a pair of equations (3.5) for an OR gate.

Example 1. Figure 3.3 represents a 7-3 counter, composed of full adders.

The following equations can be derived for this network using the FA model described above.

$$\begin{cases} x_1 + x_2 + x_3 - 2x_{11} - x_{12} = 0 \\ x_4 + x_5 + x_6 - 2x_{13} - x_{14} = 0 \\ x_{12} + x_{14} + x_7 - 2x_{15} - x_{10} = 0 \\ x_{11} + x_{13} + x_{15} - 2x_8 - x_9 = 0 \end{cases} \tag{3.6}$$

Mathematically, we can represent it as:

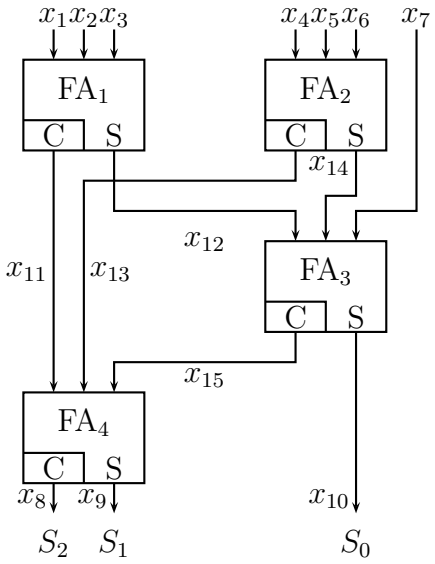


Figure 3.3. Arithmetic network of a 7-3 counter.

$$A\ x = b \tag{3.7}$$

where A is an $m \times n$ constraint matrix, x is an n-vector representing the signals and b is a constant vector. For the 7-3 counter, we have:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 1 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

We now describe a method to represent such a network by a single *algebraic signature* expression that can be used to verify the circuit against its intended function.

First, we introduce the concept of a *Reference Signature*, which provides the expected relationship between primary inputs and outputs of the arithmetic circuit, and serves as a "golden model" of the circuit.

Definition 3.2.1. *The* **Input Signature**, *or* $\mathbf{Sig_I(N)}$, *of an arithmetic circuit, N, is a linear combination of primary input variables that represents the integer function computed by the circuit.*

Example 2. For a 7-3 counter, N_{7-3}, with inputs $\{x_1,x_2,x_3,x_4,x_5,x_6,x_7\}$, the input signature is given by:

$$Sig_I(N_{7-3}) = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \tag{3.8}$$

Example 3. For a n-bit binary adder, N_A, with inputs $\{a_0, b_0, \cdots, a_{n-1}, b_{n-1}\}$, the input signature is given by:

$$Sig_I(N_A) = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i$$

Example 4. Consider a 2-bit unsigned multiplier, N_{M2}, with inputs $\{a_0, b_0, a_1, b_1\}$. Since the multiplier is a non-linear circuit, we first need to convert its primary inputs into new variables called partial product terms, pp_I as follows:

$$\begin{aligned} A \cdot B &= (2a_1 + a_0).(2b_1 + b_0) \\ &= 4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0b_0 \\ &= 4pp_3 + 2pp_2 + 2pp_1 + pp_0 \end{aligned} \tag{3.9}$$

The variables pp_i become primary inputs to our multiplier and the input signature is given by:

$$Sig_I(N_{M2}) = 4pp_3 + 2pp_2 + 2pp_1 + pp_0 \tag{3.10}$$

Mathematically, Input Signature, $Sig_I(N)$, can be represented as:

$$\boxed{Sig_I(N) = r_I^T \; x_I}$$

Where r_I is a input signature vector and x_I is a set of primary inputs.

Example 5. In the case of a 2-bit unsigned multiplier, we have:

$$r_I^T \; x_I = [4 \; 2 \; 2 \; 1] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Where $x_1 = pp_3$, $x_2 = pp_2$, $x_3 = pp_1$ and $x_4 = pp_0$.

Definition 3.2.2. *The* **Output Signature**, *or* $\mathbf{Sig_O(N)}$, *of an arithmetic circuit, N, is defined as a linear combination of the primary output signals that represent the n-bit encoding of the output word computed by the circuit.*

Example 6. For a 7-3 counter, N_{7-3}, with outputs $\{x_8, x_9, x_{10}\}$, the output signature is given by:

$$Sig_O(N_{7-3}) = 4x_8 + 2x_9 + x_{10} \tag{3.11}$$

Example 7. For a n-bit binary adder, N_A, with outputs $\{C, S_{n-1}, S_{n-2} \cdots, S_0\}$, the output signature is given by:

$$Sig_O(N_A) = 2^n \cdot C + \sum_{i=0}^{n-1} 2^i. \; S_i$$

Example 8. For a 2-bit unsigned multiplier, N_{M2}, with outputs $\{z_3, z_2, z_1, z_0\}$, the output signature is given by:

$$Sig_O(N_{M2}) = 8z_3 + 4z_2 + 2z_1 + z_0 \tag{3.12}$$

In general, an output signature for any arithmetic circuit with n outputs S_i is represented as

$$Sig_O(N) = \sum_{i=0}^{n-1} 2^i \; S_i$$

Mathematically, Output Signature, $Sig_O(N)$, can be represented as:

$$\boxed{Sig_O(N) = r_O^T \; x_O}$$

Where r_O is a output signature vector and x_O is a set of primary outputs.

Example 9. In the case of a 2-bit unsigned multiplier, we have:

$$r_O^T \; x_O = [8 \; 4 \; 2 \; 1] \begin{bmatrix} z_3 \\ z_2 \\ z_1 \\ z_0 \end{bmatrix}$$

Definition 3.2.3. *The* **Reference Signature**, *or* **Ref**(**N**), *of an arithmetic circuit is defined as the difference between its output signature and input signature and is given by:*

$$\boxed{Ref(N) = Sig_O - Sig_I} \tag{3.13}$$

Mathematically in terms of input and output signature vectors, the Reference Signature, $Ref(N)$ is defined as follows:

$$\boxed{Ref(N) = [-r_I \; r_O]^T \begin{bmatrix} x_I \\ x_O \end{bmatrix}}$$

Where r_I is a input integer signature vector, r_O is an output integer signature vector, x_I is a set of primary inputs and x_O is a set of primary outputs.

Example 10. For the 7-3 counter example, with $x_I = [x_1, \cdots, x_7]$ and $x_O = [x_8, x_9, x_{10}]$, the reference signature, Ref(N), is given by:

$$Ref(N_{7-3}) = 4x_8 + 2x_9 + x_{10} - x_1 - x_2 - x_3 - x_4 - x_5 - x_6 - x_7 \tag{3.14}$$

Or, equivalently as

$$[-r_I\ r_O]^T \begin{bmatrix} x_I \\ x_O \end{bmatrix} = [-1\ -1\ -1\ -1\ -1\ -1\ -1\ 4\ 2\ 1] \begin{bmatrix} x_I \\ x_O \end{bmatrix}$$

Here the output signature corresponds to the 3-bit encoding of the output word, while the input signature is a sum of the input bits.

In our work an arithmetic network N will be similarly represented by its *algebraic signature*, $Sig(N)$, and compared to the reference signature $Ref(N)$. For the circuit to be functionally correct, its algebraic signature must match its reference signature, i.e.,

$$Sig(N) = Ref(N) \tag{3.15}$$

Definition 3.2.4. *The* **Algebraic Signature**, *or* **Sig(N)**, *of an arithmetic network N, is given by:*

$$\boxed{Sig(N) = r^T\ x}$$

Where $r = [r_I\ r_O\ r_S]$ is an integer signature vector and $x = [x_I\ x_O\ x_S]^T$ is a set of variables representing signals.

Our goal is to compute an algebraic signature of a network as a means of proving its functionality expressed by the reference signature. This can be accomplished by setting up a linear system describing the arithmetic network and solving it using a linear programming (LP) solver described in next section.

CHAPTER 4

MATHEMATICAL FORMULATION OF FUNCTIONAL VERIFICATION PROBLEM

This section describes the method for computing algebraic signature for arithmetic networks using linear algebra techniques and discusses the issue of signature matching.

4.1 Mathematical Formulation

Let n be the total number of signals in the network, each represented by a variable x_k, and m be the number of linear equations in the system. The network is then represented in matrix form as

$$A\ x = b \tag{4.1}$$

Where A is an $m \times n$ matrix representing the network, x is an n-vector representing the signals, and b is a constant vector of size m.

The vector x of signal variables is further partitioned into a set of input signals x_I, output signals x_O, and internal signals x_S. Matrix A is similarly partitioned into submatrices A_I, A_O, A_S. Then, the system $Ax = b$ can be written as:

$$A_I x_I + A_O x_O + A_S x_S = b$$

Where A_I is the matrix A restricted to the columns associated with primary input signals x_I, A_O is the matrix A restricted to the columns of primary output signals x_O and A_S is the matrix A restricted to the columns of internal signals x_S.

For the 7-3 counter of Fig. 3.3, $x^T = [x_1, \ldots, x_{15}]$, where $x_I^T = [x_1, x_2, x_3, x_4, x_5, x_6, x_7]$, $x_O^T = [x_8, x_9, x_{10}]$, $x_S^T = [x_{11}, x_{12}, x_{13}, x_{14}, x_{15}]$, $n = 15$, $m = 4$ and $b = 0$.

Mathematical representation of the network is given as follows:

$$\begin{array}{c} \\ \text{A} = \end{array} \begin{array}{c} \begin{array}{ccccccc|ccc|ccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \end{array} \\ \left(\begin{array}{ccccccc|ccc|ccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 1 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right) \end{array} \text{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Given an arithmetic network, represented by $A\ x = b$, the system computes its algebraic signature vector $r = [-r_I, r_O, r_S]$, in an attempt to match the reference signature vector $[-r_I, r_O]$. This is done by combining the rows of A into a single *algebraic signature* $Sig(N) = r^T\ x$, by finding a linear combination α of rows of A that produces $r = [-r_I, r_O, r_S]$. This operation is similar to a classical Gaussian elimination. Intuitively, since we want to eliminate all the internal signals from the algebraic signature, we want to have $r_S = 0$. By doing this, only primary input and output signals will be involved in the signature. It will be shown later that $r_S = 0$ is sufficient but not a necessary condition for satisfying the reference signature.

Recalling that matrix A and variables x are partitioned into groups associated with inputs, outputs, and internal signals, the signature $Sig(N) = r^T\ x$ for network N is obtained from a signature vector, r, computed as follows:

$$r = A^T\alpha \Leftrightarrow \begin{cases} A_I^T\alpha = -r_I \\ A_O^T\alpha = r_O \\ A_S^T\alpha = r_S \end{cases} \tag{4.2}$$

Here α is a vector of integer coefficients (to be computed) that reduces linear equations defined by matrix A to the signature vector r. Vectors r_O and r_I are provided by the

user as part of the reference signature to be verified. Since the goal is to match the algebraic signature with the reference, we need to find vector α such that

$$[A_I, A_O]^T \ \alpha = \begin{bmatrix} -r_I \\ \\ r_O \end{bmatrix} \tag{4.3}$$

Such a computation can be done easily and efficiently using any linear programming (LP) system. The LP system solves these equations in terms of α as a constraint satisfaction problem, with no objective function. Basically, searching for α is searching for a linear combination of the rows of matrix A that results in the algebraic signature for the network.This is typically done using Simplex algorithm during the pre-solving phase of any LP system, which is fast and scalable.

Note that no integer constraints need to be imposed on variables x since we are solving for α and not for x. This is the main difference between our approach and all the known approaches based on LP or ILP proposed so far, making this computation fast, efficient and highly scalable.

4.2 Computing the Signature

Given the network, described by $A\,x = b$, and its reference signature $[-r_I, r_O] \begin{bmatrix} x_I \\ \\ x_O \end{bmatrix}$, its algebraic signature, $r^T\ x$, is computed with the following two-step approach. First we attempt to solve the system described by equations,

$$\begin{cases} A_I^T \alpha = -r_I \\ A_O^T \alpha = r_O \\ A_S^T \alpha = 0 \end{cases}$$

with given r_I, r_O and with r_S forced to 0. If the solution to this system exists, there exists α for which these equations hold, the reference signature is valid and the

circuit is proved to be correct. In this case resulting vector α provides the desired linear combination of the rows of A that reduces it to the reference signature without any internal signals, x_S.

If the system is infeasible, we relax the constraint on r_S so that r_S is no longer required to be 0 and solve a reduced system in which r_S is allowed to take arbitrary value. We refer to this process as "Completing the Signature" (in terms of r_S), i.e., we solve the following linear system for α using standard LP solver:

$$\begin{cases} A_I^T \ \alpha = -r_I \\ A_O^T \ \alpha = r_O \end{cases} \tag{4.4}$$

Note that the vector r_S associated with internal variables x_S is not being used in the computation. If the system still has no solution, i.e, there is no linear combination of rows of A that will produce the algebraic signature that matches the reference signature vector $[-r_I, r_O]$, the circuit is incorrect. If the system has solution, α, the signature vector r_S associated with the internal variables is computed as follows:

$$r_S = A_S^T \ \alpha$$

Ideally, we would like to have all the internal variables eliminated from the algebraic signature, i.e., $r_S = 0$, as a condition for satisfying the reference signature. However, this may not always be possible with a basic system of equality constraints, $A\ x = b$, and additional constraints of Boolean nature may need to be imposed. This issue, and the consequences of non-zero r_S, will be illustrated by the following examples.

Let us revisit the 7-3 counter circuit described by equation (3.6) with the reference signature given by equation (3.14). Its input and output reference signature vectors are given by:

$$r_I^T = [-1 \ -1 \ -1 \ -1 \ -1 \ -1 \ -1]$$
$$r_O^T = [4 \ 2 \ 1]$$

According to our procedure, we first force r_S to zero and compute α using the following relationship:

$$[A_I,\ A_O,\ A_S]\ \alpha = \begin{bmatrix} -r_I \\ r_O \\ 0 \end{bmatrix}$$

The system has the following solution:

$$\alpha^T = [-1\ \ -1\ \ -1\ \ -2]$$

Subsequently, the following signature is computed for this system:

$$Sig(N_{7-3}) = r^T\ x = -x_1 - x_2 - x_3 - x_4 - x_5 - x_6 - x_7 + 4x_8 + 2x_9 + x_{10}.$$

The internal signals $x_S = [x_{11}, x_{12}, x_{13}, x_{14}, x_{15}]$ have been completely eliminated, so that the computed signature matches the reference signature and the design is considered correct.

4.3 Residual Expression

An interesting question arises, what if the computed signature $Sig(N)$, in addition to the reference signature, contains an expression associated with the internal signals, i.e., if $r_S \neq 0$. We refer to such an expression as a *Residual Expression*, denoted $RE(N)$:

$$RE(N) = Sig(N) - Ref(N) = r_S^T\ x_S \tag{4.5}$$

Does the existence of a non-empty residual expression, $RE(N) \neq \emptyset$ imply that the system does not satisfy the reference signature and the design is incorrect? It turns out that it is not necessarily the case, and that $r_S = 0$ is a *sufficient but not a necessary* condition for the circuit to be correct. It will be shown in Chapter 5 that as long

as $RE(N)$ evaluates to zero for all values of internal signal variables x_S produced by the network, the network signature will match the reference signature and the design will be correct. This case is illustrated with the following example.

Example 1: Consider a signed 2×2 multiplier network, N_{M2}, shown in Figure 4.1.

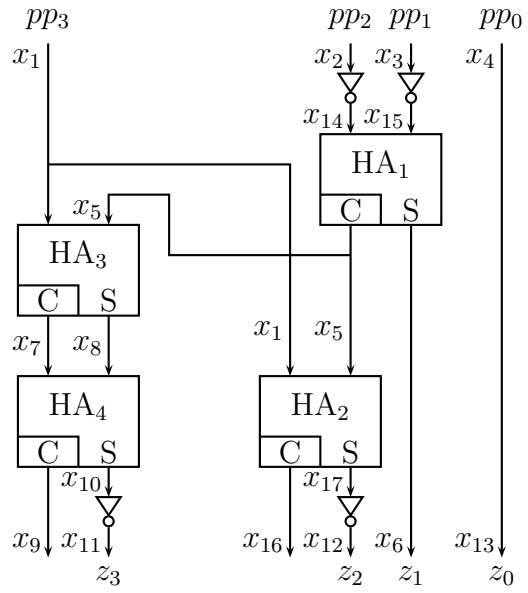


Figure 4.1. Signed 2×2 multiplier network.

The network is described by the following equations:

$$\begin{cases} x_{14} + x_{15} - x_6 - 2x_5 = 0 \\ x_1 + x_5 - x_{17} - 2x_{16} = 0 \\ x_7 + x_8 - x_{10} - 2x_9 = 0 \\ x_1 + x_5 - x_8 - 2x_7 = 0 \\ x_2 + x_{14} = 1 \\ x_3 + x_{15} = 1 \\ x_4 - x_{13} = 0 \\ x_{17} + x_{12} = 1 \\ x_{10} + x_{11} = 1 \end{cases} \tag{4.6}$$

or, equivalently in matrix $A\ x = b$ with:

$$
\begin{array}{c}
\\
\mathrm{A} = \\
\\
\end{array}
\begin{array}{c}
\begin{array}{cccc|cccc|cccccccccc}
x_1 & x_2 & x_3 & x_4 & x_{11} & x_{12} & x_6 & x_{13} & x_5 & x_7 & x_8 & x_9 & x_{10} & x_{14} & x_{15} & x_{16} & x_{17}
\end{array} \\
\left(\begin{array}{cccc|cccc|cccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -2 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -2 & -1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}\right)
\end{array}
\quad \mathrm{b} = \quad
\left(\begin{array}{c}
0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1
\end{array}\right)
$$

Here $x_1 = pp_3$, $x_2 = pp_2$, $x_3 = pp_1$, $x_4 = pp_0$ and $x_{11} = z_3$, $x_{12} = z_2$, $x_6 = z_1$, $x_{13} = z_0$. The combination of HA_3 and HA_4 models an OR gate (see equation 3.5) that appears in this network. Inputs to the network are partial product terms $\{pp_0, pp_1, pp_2, pp_3\}$, generated by a partial product generator module, from the actual inputs of the multiplier, a_1, a_0, b_1, b_0. Hence, the expected input signature, $Sig_I(N_{M2})$, for the network is:

$$
\begin{aligned}
Sig_I(N_{M2}) &= (-2a_1 + a_0)(-2b_1 + b_0) \\
&= 4a_1b_1 - 2a_1b_0 - 2a_0b_1 + a_0b_0 \\
&= 4pp_3 - 2pp_2 - 2pp_1 + pp_0
\end{aligned}
\tag{4.7}
$$

The output signature is obtained directly from the encoding of the output bits, $Sig_O(N_{M2}) = -8z_3 + 4z_2 + 2z_1 + z_0$, so the reference signature for this design is:

$$
[r_O,\ -r_I]^T \begin{bmatrix} x_O \\ x_I \end{bmatrix} = Ref(N_{M2}) = -8z_3 + 4z_2 + 2z_1 + z_0 - 4pp_3 + 2pp_2 + 2pp_1 - pp_0
$$

In order to compute an algebraic signature, first we try to solve the system for α by forcing r_S to zero using the following relationship:

$$A^T\ \alpha = \begin{bmatrix} -r_I \\ r_O \\ 0 \end{bmatrix}$$

In this case there is no feasible solution to the system. Therefore, we try to complete the signature in-terms of r_S. Subsequently we must consider r_S to be a free signal and find α using the following relationship:

$$[A_I,\ A_O]^T\ \alpha = \begin{bmatrix} -r_I \\ \\ r_O \end{bmatrix}$$

This system has a solution, $\alpha^T = [-2\ -4\ -8\ 0\ -8\ 4\ 1\ 2\ 2]$ and the corresponding algebraic signature computed by the system is:

$$Sig(N_{M2}) = -4pp_3 + 2pp_2 + 2pp_1 - pp_0 - 8z_3 + 4z_2 + 2z_1 + z_0 \\ +8x_7 + 4x_8 - 8x_{10} + 4x_{17} \tag{4.8}$$

In this case $Sig(N) - Ref(N) \neq \emptyset$, and has a residual expression,

$$RE(N_{M2}) = r_S^T\ x_S = -8x_7 - 4x_8 + 8x_{10} - 4x_{17}$$

associated with the internal signals. Does this mean that our system does not satisfy the reference signature and the design is incorrect? It turns out that this expression actually evaluates to zero with the help of additional constraints, to be discussed in the next section.

In summary, we have seen two cases:

- When $r_S = 0$, all the internal signals are automatically eliminated from the input/output relationship

- When $r_S \neq 0$, some internal signals. are present but the residual expression can be reduced to zero for all values of the internal variables that can be produced by the system which will be discussed in the next section.

CHAPTER 5

THEOREMS AND BOOLEAN MODELS

In this section we will present a set of theorems that govern our approach to functional verification of arithmetic circuits. In the following, the circuit is considered correct if it satisfies the functionality expressed by the reference signature i.e., if its algebraic signature matches the reference signature. We also discuss the need for additional constraints to properly model the inherent Boolean nature of arithmetic circuits.

5.1 Theorems

Theorem 5.1.1. *Given an arithmetic circuit N represented by $A\ x = b$ with reference signature $Ref(N) = [-r_I, r_O]^T \begin{bmatrix} x_I \\ x_O \end{bmatrix}$, the ciruit is incorrect if there is no solution to equation,*

$$[A_I, A_O]^T\ \alpha = \begin{bmatrix} -r_I \\ r_O \end{bmatrix}$$

Proof. The lack of solution to this equation indicates that the coefficients associated with the input and output variables do not match those of the reference signature. Hence the circuit does not satisfy the reference signature and by definition it is *incorrect*. □

The remaining theorems consider the case when the computed algebraic signature contains residual expression, $RE(N)$. We distinguish two cases:

- When there is no residual expression ($RE = \phi$) in the algebraic signature, i.e. when $r_S = 0$; and
- When $RE(N)$ is not empty ($r_S \neq 0$) but $RE(N) = r_S^T\ x_S$ evaluates to zero for all possible values of internal variables x_S produced by the circuit.

Theorem 5.1.2. *Given an arithmetic circuit N represented by $A\ x = b$ with reference signature $Ref(N) = [-r_I, r_O]^T \begin{bmatrix} x_I \\ x_O \end{bmatrix}$, the circuit is correct if its computed algebraic signature matches exactly the reference signature and contains no residual expression $RE(N)$, i.e., $r_S = 0$. That is $RE(N) = \phi$ is a sufficient condition for the arithmetic circuit to be correct.*

Proof. By definition, the Residual Expression, $RE(N)$, is the difference between its algebraic signature ($r^T\ x$) and reference signature, $[-r_I, r_O]^T \begin{bmatrix} x_I \\ x_O \end{bmatrix}$, that is

$$RE(N) = r^T\ x - [-r_I, r_O]^T \begin{bmatrix} x_I \\ x_O \end{bmatrix}$$

Since

$$r^T\ x = [-r_I, r_O, r_S]^T \begin{bmatrix} x_I \\ x_O \\ x_S \end{bmatrix}$$

We have

$$RE(N) = [-r_I, r_O, r_S]^T \begin{bmatrix} x_I \\ x_O \\ x_S \end{bmatrix} - [-r_I, r_O]^T \begin{bmatrix} x_I \\ x_O \end{bmatrix} = r_S^T\ x_S$$

If the circuit has no residual expression, $RE(N) = \phi$, then

$$RE(N) = r_S^T \; x_S = 0$$

which implies that $r_S = 0$. That is, no internal signal variables are involved in the signature and hence the algebraic and reference signatures match indicating that the circuit is correct. □

Example: To illustrate this case, consider again a (7-3) counter in Figure 5.1.

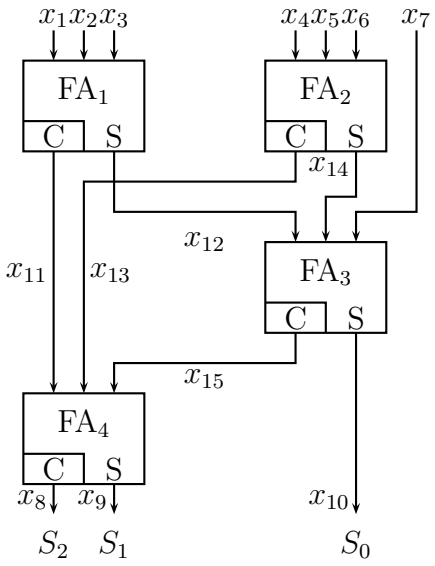


Figure 5.1. Arithmetic network of a (7-3) counter.

The network has the following sets of variables, $x_I^T = [x_1, x_2, x_3, x_4, x_5, x_6, x_7]$, $x_O^T = [x_8, x_9, x_{10}]$, $x_S^T = [x_{11}, x_{12}, x_{13}, x_{14}, x_{15}]$ and can be described by a set of linear equations 3.6, repeated here for reference.

$$\begin{cases} x_1 + x_2 + x_3 - 2x_{11} - x_{12} = 0 \\ x_4 + x_5 + x_6 - 2x_{13} - x_{14} = 0 \\ x_{12} + x_{14} + x_7 - 2x_{15} - x_{10} = 0 \\ x_{11} + x_{13} + x_{15} - 2x_8 - x_9 = 0 \end{cases}$$

The reference signature is given by:

$$Ref(N) = 4x_8 + 2x_9 + x_{10} - (x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7) \tag{5.1}$$

ans reference signature vectors are given by:

$$-r_I^T = [1\ 1\ 1\ 1\ 1\ 1\ 1]$$
$$r_O^T = [4\ 2\ 1]$$

Computation of the signature using the linear system

$$[A_I,\ A_O,\ A_S]^T\ \alpha = \begin{bmatrix} -r_I \\ r_O \\ 0 \end{bmatrix}$$

gives the following solution:

$$\alpha^T = [-1\ \ -1\ \ -1\ \ -2]$$

That is, there exists a linear combination α of rows of A whose sum reduces to a reference signature without any internal signals ($r_S = 0$). The resulting algebraic signature is:

$$r^T\ x = 4x_8 + 2x_9 + x_{10} - x_1 - x_2 - x_3 - x_4 - x_5 - x_6 - x_7$$

The signature is identical with the reference signature of Equation 5.1 and does not contain any of the internal signals, x_s, i.e., $r_s = 0$. Hence the design is correct.

A stronger, necessary and sufficient condition for circuit correctness is that $RE(N) = 0$, i.e., when a possibly non-empty $RE(N) = r_S^T\ x_S$, with $r_S \neq 0$, evaluates to zero for

all possible values of x_S generated by the circuit. The fact that $RE(N) \neq \phi$ does not necessarily mean that the circuit is *incorrect*. The numerical value of $RE(N)$ must be proved to be zero.

Theorem 5.1.3. *Given an arithmetic circuit N represented by $A\ x = b$ with reference signature $Ref(N) = [-r_I, r_O]^T \begin{bmatrix} x_I \\ x_O \end{bmatrix}$, the circuit is correct iff $RE(N) = r_S^T\ x_S = 0$ i.e., it evaluates to zero for all the values of x_S produced by the network.*

Proof. As previously shown, the residual expression $RE(N)$ can be written as

$$RE(N) = [-r_I, r_O, r_S]^T \begin{bmatrix} x_I \\ x_O \\ x_S \end{bmatrix} - [-r_I, r_O]^T \begin{bmatrix} x_I \\ x_O \end{bmatrix} = r_S^T\ x_S$$

If the circuit has non-empty residual expression, the vector $r_S \neq 0$ and the only way $RE(N)$ can evaluate to zero is that the linear combination of variables x_S evaluates to zero. If this is the case, $RE(N) = 0$ and $Sig(N) = Ref(N)$. If the residual expression evaluates to zero for all values of x_S, then $RE(N) = r_S^T\ x_S = 0$ and $Sig(N) = Ref(N)$, i.e., the circuit is correct.

Now, let's assume that the circuit is correct, and show that it implies that $RE(N) = 0$. Again, recall that $Sig(N) - Ref(N) = RE(N)$. If the circuit is correct, then $Sig(N) = Ref(N)$ and $RE(N) = 0$ □

Note that Theorem 5.1.3 expresses a more general case than the one stated in Theorem 5.1.2 for $RE(N) = \phi$. This is because $RE(N) = r_S^T\ x_S = 0$ can be obtained by either having $r_S = 0$ (empty $RE(N)$) or, for non-zero r_S, having the linear combination $r_S^T\ x_S$ evaluate to zero. This theorem points out to an interesting difficulty in proving correctness of arithmetic circuit in cases where RE is non-empty. Additional constraints must be imposed on the circuit in an attempt to simplify linear equations or to prove $RE = 0$ directly.

The following section explains this issue.

5.2 Imposing Binary Constraints

Proving that $RE = 0$ for all the values of internal signals poses a new problem that requires additional insight into a binary nature of the network. This can be done by examining the circuit and deriving additional constraints, in form of Boolean invariants and assertions. Deriving such invariants is done routinely and efficiently by logic synthesis and verification tools, such as ABC [22, 5]. However, our initial experience shows that the problem may be simpler as it relates to a special class of designs, namely integer arithmetic circuits. These constraints are of binary nature and can be viewed as special cases of Boolean constraints needed to properly model the integer arithmetic semantic of the network. Our initial analysis shows that these constraints can be classified as follows:

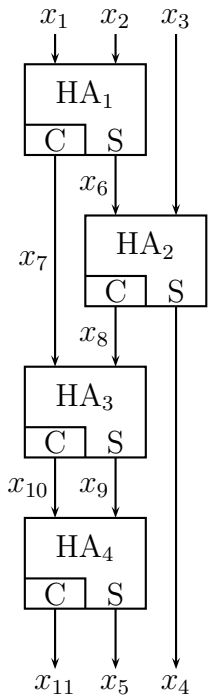
• *Structural constraints*, caused by fanout of internal signals; these can be easily modeled as equality constraints on the corresponding signals (such as $x_8 = x_{17}$ in Example 2);

• *Boolean constants* that can be propagated in the network, such as the C output of two half-adders connected in series (e.g., signal x_9 in Example 2, Figure 4.1); and

• *Binary constraints* that may result from specific network configurations, such as internal signals of a gate-level implementation of a full-adder. In some cases, such as those for multipliers shown in Table 7.4, they can be derived relatively easily, but in general cases the problem may require more work.

5.3 Deriving Boolean Constraints

We first demonstrate how to derive Boolean constraints in a HA network shown in Figure 5.2. Such a network often appears in arithmetic circuits composed of wallace trees. We will show that this network represents a full adder (FA) circuit and will



$$\begin{cases} x_1 + x_2 - 2x_7 - x_6 = 0 \\ x_3 + x_6 - 2x_8 - x_4 = 0 \\ x_7 + x_8 - 2x_{10} - x_9 = 0 \\ x_9 + x_{10} - 2x_{11} - x_5 = 0 \end{cases} \tag{5.2}$$

Figure 5.2. Configuration in Netlist.

derive the constraints on some of the internal signals resulting from this configuration. This will provide important Boolean information necessary to correctly model arithmetic circuit as a system of algebraic equations.

First we prove that $x_{11} = 0$. The C and S outputs of a HA (in this case HA_3) are never equal to 1 at the same time, i.e., $C \cdot S = 0$. Since $x_{11} = x_{10} \cdot x_9$, we have $x_{11} = 0$. Hence, the equations 5.2 can be rewritten as

$$\begin{cases} x_1 + x_2 - 2x_7 - x_6 = 0 \\ x_3 + x_6 - 2x_8 - x_4 = 0 \\ x_7 + x_8 - 2x_{10} - x_9 = 0 \\ x_9 + x_{10} - x_5 = 0 \end{cases}$$

Next, we will show that $x_{10} = 0$ by considering the following Boolean equations, where $\wedge$ represents AND, $\vee$ represents OR and $\oplus$ represents XOR.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0
0	1	0	1	0	1	0	0	0	0	0
0	1	1	0	1	1	0	1	1	0	0
1	0	0	1	0	1	0	0	0	0	0
1	0	1	0	1	1	0	1	1	0	0
1	1	0	0	1	0	1	0	1	0	0
1	1	1	1	1	0	1	0	1	0	0

Table 5.1. Truth table for full-adder circuit in Figure 5.2(a)

$$x_7 = x_1 \wedge x_2$$

$$x_6 = x_1 \oplus x_2$$

$$x_8 = x_6 \wedge x_3 = (x_1 \oplus x_2) \wedge x_3$$

$$x_{10} = x_7 \wedge x_8 = (x_1 \wedge x_2) \wedge ((x_1 \oplus x_2) \wedge x_3) = 0$$

Since $x_{10} = 0$, the equation

$$x_7 + x_8 - 2x_{10} - x_9 = 0$$

simplifies to

$$x_7 + x_8 - x_9 = 0$$

And with $x_{11} = 0$, we have

$$x_5 = x_9 = x_7 + x_8$$

As a result HA_3 and HA_4 can be replaced by a simplified HA such that $x_5 = x_7 + x_8$. The modified equations clearly indicate that the circuit in Figure 5.2(a) forms a Full-adder (FA):

$$x_1 + x_2 + x_3 = 2x_5 + x_4$$

The proof that this network represents a full adder can also be done by computing

its algebraic signature, namely in Figure 5.2

$$
\begin{aligned}
& x_1 + x_2 + x_3 \\
& = 2x_7 + (x_6 + x_3) \\
& = 2x_7 + 2x_8 + x_4 \\
& = 2(x_7 + x_8) + x_4 \\
& = 2x_5 + x_4
\end{aligned}
$$

As we will see later, the constraint that the internal OR variable (x_{10}) is zero in such a configuration is essential in simplifying linear equations of the network to prove its functional correctness.

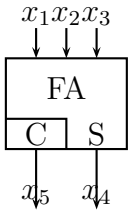


Figure 5.3. Full Adder symbol : $x_1 + x_2 + x_3 = 2x_5 + x_4$

The proof that $x_{10} = 0$ can be also shown using algebraic equations, rather than Boolean, as follows:

$$
\begin{aligned}
& x_6 = x_1 + x_2 - 2x_1 \cdot x_2 \\
& x_7 = x_1 \cdot x_2 \\
& x_8 = x_6 \cdot x_3 = x_1 \cdot x_3 + x_2 \cdot x_3 - 2x_1 \cdot x_2 \cdot x_3 \\
& x_{10} = x_7 \cdot x_8 \\
& = x_1 \cdot x_2(x_1 \cdot x_3 + x_2 \cdot x_3 - 2x_1 \cdot x_2 \cdot x_3) = x_1 \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3 - 2x_1 \cdot x_2 \cdot x_3 = 0
\end{aligned}
$$

The above simplification is true for Boolean variables.

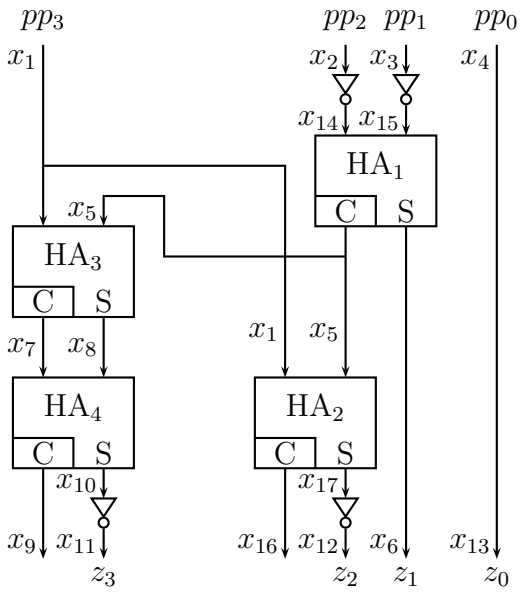


Figure 5.4. Signed 2×2 multiplier network.

Example 1: Let us revisit the signed 2×2 multiplier network, shown in Figure 4.1.

The network is described by set of linear equations 4.6, repeated here for reference.

$$\begin{cases} x_{14} + x_{15} - x_6 - 2x_5 = 0 \\ x_1 + x_5 - x_{17} - 2x_{16} = 0 \\ x_7 + x_8 - x_{10} - 2x_9 = 0 \\ x_1 + x_5 - x_8 - 2x_7 = 0 \\ x_2 + x_{14} = 1 \\ x_3 + x_{15} = 1 \\ x_4 - x_{13} = 0 \\ x_{17} + x_{12} = 1 \\ x_{10} + x_{11} = 1 \end{cases}$$

The partial product terms $\{pp_0, pp_1, pp_2, pp_3\}$ are generated from the actual inputs of the multiplier, a_1, a_0, b_1, b_0, by a partial product generator module.

The reference signature is given by

$$[r_O,\ r_I]^T \begin{bmatrix} -x_I \\ x_O \end{bmatrix} = -8x_{11} + 4x_{12} + 2x_6 + x_{13} - 4x_1 + 2x_2 + 2x_3 - x_4$$

As before, we try to solve the system for α by forcing r_S to zero. In this case, the system has no solution. So we use the following equation to compute the signature

$$[A_I,\ A_O]^T\ \alpha = \begin{bmatrix} -r_I \\ r_O \end{bmatrix}$$

This system has solution, $\alpha^T = [-2\ \ -4\ 0\ 0\ \ -8\ 4\ \ -1\ 2\ 2]$ and the corresponding algebraic signature is:

$$r^T\ x = -8x_{11} + 4x_{12} + 2x_6 + x_{13} - 4x_1 + 2x_2 + 2x_3 - x_4 \\ +8x_7 + 4x_8 - 8x_{10} + 4x_{17} \tag{5.3}$$

Note that in this case the algebraic signature has a non-empty residual expression:

$$r_S^T\ x_S = -8x_7 - 4x_8 + 8x_{10} - 4x_{17} \tag{5.4}$$

associated with the internal signals. According to Theorem 5.1.3, in order to prove that the design is correct we must check if the residual expression evaluates to zero for all possible values of x_7, x_8, x_{17} and x_{10} produced by the network. To do that, note that signals x_8, x_{17} are equivalent outputs S of HA_2 and HA_3 that share the same inputs, i.e., $x_8 = x_{17}$. By substituting this relation into equation 5.4, we obtain:

$$r_S^T\ x_S = -8x_7 - 8x_8 + 8x_{10} \tag{5.5}$$

Recall that $x_9 = 0$ since this is the output of second HA in the OR configuration. The C output of the second HA modeling the OR gates is always zero as proven earlier (Figure 5.2(a)). Hence equation

$$x_7 + x_8 - x_{10} - 2x_9 = 0$$

simpliefies to

$$x_7 + x_8 - x_{10} = 0$$

Substituting this relation into equation 5.5, we obtain:

$$r_S^T \; x_S = -8x_{10} + 8x_{10} = 0$$

With the residual expression equal to zero the algebraic signature is equal to the reference signature, proving that the design is correct.

There is another way to prove this using result of Theorem 5.1.2. Namely, add the constraints

$$x_9 = 0$$

$$x_8 = x_{17}$$

$$x_7 = x_{16}$$

to $A\,x = b$. The new system has solution, $\alpha^T = [-2\ 0\ -8\ -4\ -8\ 4\ -1\ 2\ 2\ -16\ 8\ 8]$ and the corresponding algebraic signature is:

$$r^T \; x = -8x_{11} + 4x_{12} + 2x_6 + x_{13} - 4x_1 + 2x_2 + 2x_3 - x_4$$

This signature is identical to the reference signature and it can be seen that there is no residual expression proving that the design is correct.

Example 2: Consider a signed 3×3 multiplier network, N_{M3}, shown in Figure 5.6. The network is described by the following equations:

$$
\begin{cases}
x_2 = 1 - x_1 \\
x_4 = 1 - x_3 \\
x_6 = 1 - x_5 \\
x_8 = 1 - x_7 \\
x_9 + x_{10} = 2 * x_{12} + x_{11} \\
x_1 + x_{13} = 2 * x_{15} + x_{14} \\
x_{12} + x_{14} = 2 * x_{17} + x_{16} \\
x_{17} + x_{15} = 2 * x_{19} + x_{18} \\
x_{18} + x_{19} = x_{20} \\
x_3 + x_{16} = 2 * x_{22} + x_{21} \\
x_{22} + x_5 = 2 * x_{24} + x_{23} \\
x_{20} + x_{23} = 2 * x_{26} + x_{25} \\
x_{26} + x_{24} = 2 * x_{28} + x_{27} \\
x_{27} + x_{28} = x_{29} \\
x_{25} + x_7 = 2 * x_{31} + (1 - x_{30}) \\
x_{25} + x_7 = 2 * x_{33} + x_{32} \\
x_{32} + x_{33} = x_{34} \\
x_{34} + x_{35} = 2 * x_{37} + x_{36} \\
x_{29} + x_{36} = 2 * x_{39} + x_{38} \\
x_{39} + x_{37} = 2 * x_{41} + x_{40} \\
x_{40} + x_{41} = x_{42} \\
x_{42} = 1 - x_{43} \\
x_{45} = x_{44} \\
x_{11} = x_{46} \\
x_{21} = x_{47}
\end{cases}
$$

$$\begin{cases} x_{30} = x_{48} \\ x_{38} = x_{49} \\ x_{43} = x_{50} \end{cases} \tag{5.6}$$

Inputs to the network are partial product terms $\{x_2, x_4, x_6, x_8, x_9, x_{10}, x_{13}, x_{35}, x_{45}\}$,

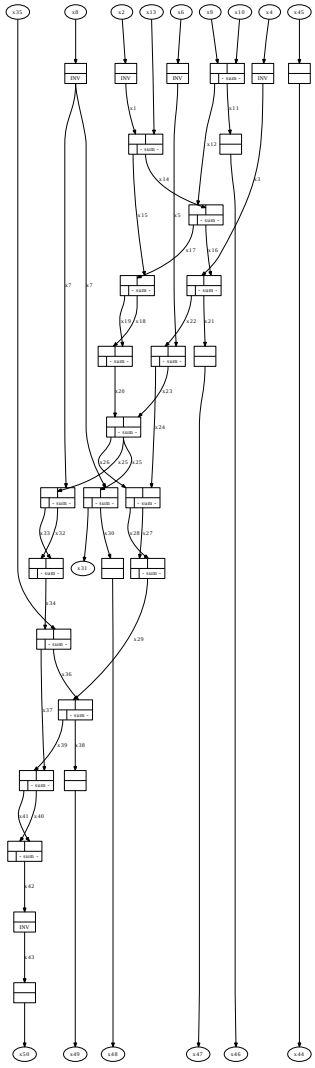


Figure 5.5. Signed 3×3 multiplier network.

generated by a partial product generator module, from the actual inputs of the multiplier, $a_2, a_1, a_0, b_2, b_1, b_0$. Hence, the expected input signature, $Sig_I(N_{M3})$, for the network is:

$$
\begin{aligned}
&Sig_I(N_{M3}) = (-4a_2 + 2a_1 + a_0)(-4b_2 + 2b_1 + b_0) \\
&= 16a_2b_2 - 8a_2b_1 - 4a_2b_0 - 8a_1b_2 + 4a_1b_1 + 2a_1b_0 - 4a_0b_2 + 2a_0b_1 + a_0b_0 \\
&= 16x_{35} - 8x_8 - 4x_4 - 8x_6 + 4x_{13} + 2x_{10} - 4x_2 + 2x_9 + x_{45}
\end{aligned} \tag{5.7}
$$

The output signature is obtained directly from the encoding of the output bits, $Sig_O(N_{M3}) = x_{44} + 2x_{46} + 4x_{47} + 8x_{48} + 16x_{49} - 32x_{50}$, so the reference signature for this design is:

$$
\begin{aligned}
&Ref(N_{M3}) = x_{44} + 2x_{46} + 4x_{47} + 8x_{48} + 16x_{49} - 32x_{50} \\
&-16x_{35} + 8x_8 + 4x_4 + 8x_6 - 4x_{13} - 2x_{10} + 4x_2 - 2x_9 - x_{45}
\end{aligned}
$$

As before, we first try to solve the system for α by forcing r_S to zero. In this case, the system has no solution. So we relax r_S and use the following equation to compute the signature

$$
[A_I,\ A_O]^T\ \alpha = \begin{bmatrix} -r_I \\ \\ r_O \end{bmatrix}
$$

This system has solution and the corresponding algebraic signature computed by the system is:

$$
\begin{aligned}
&Sig(N_{M3}) = x_{44} + 2x_{46} + 4x_{47} + 8x_{48} + 16x_{49} - 32x_{50} - 16x_{35} \\
&+8x_8 + 4x_4 + 8x_6 - 4x_{13} - 2x_{10} + 4x_2 - 2x_9 - x_{45} + 4x_3 + 8x_5 + 8x_7 - 4x_{21} \\
&-8x_{30} - 16x_{38} + 32x_{43} + 4x_{12} + 4x_{14} + 8x_{15} - 16x_{34} + 16x_{36} + 32x_{37}
\end{aligned} \tag{5.8}
$$

In this case $Sig(N) - Ref(N) \neq \emptyset$, and has a residual expression,

$$
\begin{aligned}
&RE(N_{M3}) = -4x_3 - 8x_5 - 8x_7 + 4x_{21} + 8x_{30} + 16x_{38} - 32x_{43} - 4x_{12} \\
&-4x_{14} - 8x_{15} + 16x_{34} - 16x_{36} - 32x_{37}
\end{aligned}
$$

associated with the internal signals. It can be shown that this expression actually evaluates to zero, for all possible values of x_S produced by the network. This can be

proved by adding some additional constraints (Boolean invariants) associated with the network. In particular, find that variables x_{19}, x_{28}, x_{41} are actually zero for all possible values produced by the network. The proof for this was given in Section 5.2. We also verified this by generating a BDD for this network and found that the variables x_{19}, x_{28}, x_{41} are zero as shown in Figure 5.6

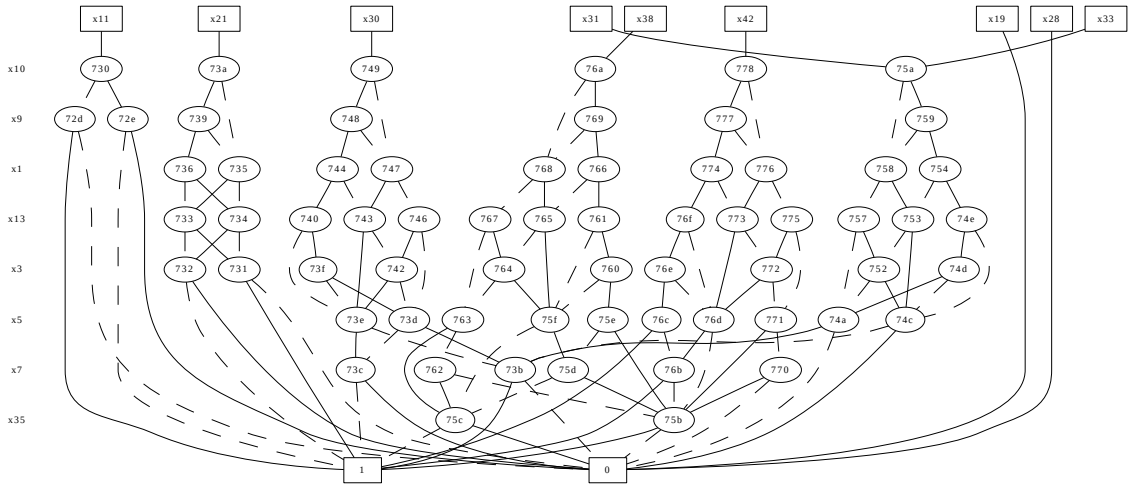


Figure 5.6. BDD of a signed 3×3 multiplier network.

It can also be seen from the network that signals x_{31}, x_{33} are equivalent outputs C of the HAs that share the same inputs i.e., x_7 and x_{25} and the S outputs of these HAs are just inverted, i.e.,

$$x_{32} = 1 - x_{30}$$

Now that we know that some of the variables are zero for all possible values of x_S produced by the network, we can now add the corresponding constraints to the network.

$$x_{19} = 0$$

$$x_{28} = 0$$

$$x_{41} = 0$$

$$x_{31} = x_{33}$$

$$x_{32} = 1 - x_{30}$$

This system has solution and the corresponding algebraic signature is

$$Sig(N_{M3}) = x_{44} + 2x_{46} + 4x_{47} + 8x_{48} + 16x_{49} - 32x_{50}$$
$$-16x_{35} + 8x_8 + 4x_4 + 8x_6 - 4x_{13} - 2x_{10} + 4x_2 - 2x_9 - x_{45}$$

It can be seen that there is no residual expression and the algebraic signature is equal to the provided reference signature, proving that the design is correct.

Example 3: Consider the example of a parallel prefix adder, N_{A4}, shown in Figure 5.7. This is a gate level implementation of a 4-bit parallel prefix adder. The arithmetic

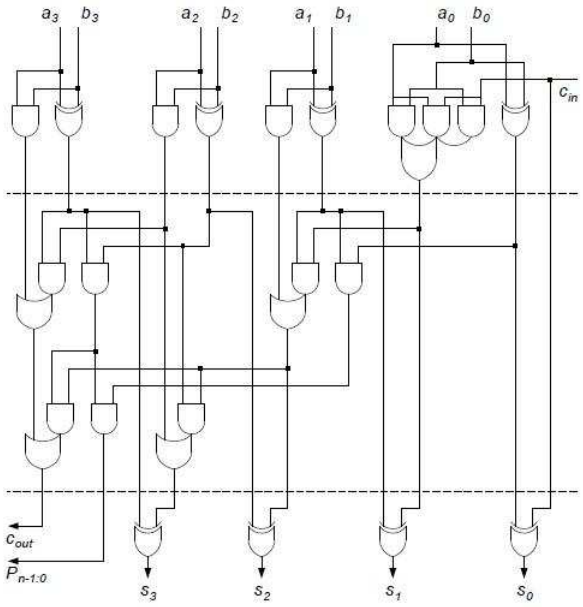


Figure 5.7. Gate level implementation of parallel prefix adder (courtesy of [52])

network in terms of half-adders is shown in Figure 5.8. Primary inputs to the network are $a_o, a_1, a_2, a_3, b_0, b_1, b_2, b_3, c_{in}$. Hence the expected input signature, Sig_I, for the network is:

$$Sig_I(N_{A4}) = a_o + 2a_1 + 4a_2 + 8a_3 + b_0 + 2b_1 + 4b_2 + 8b_3 + c_{in}$$

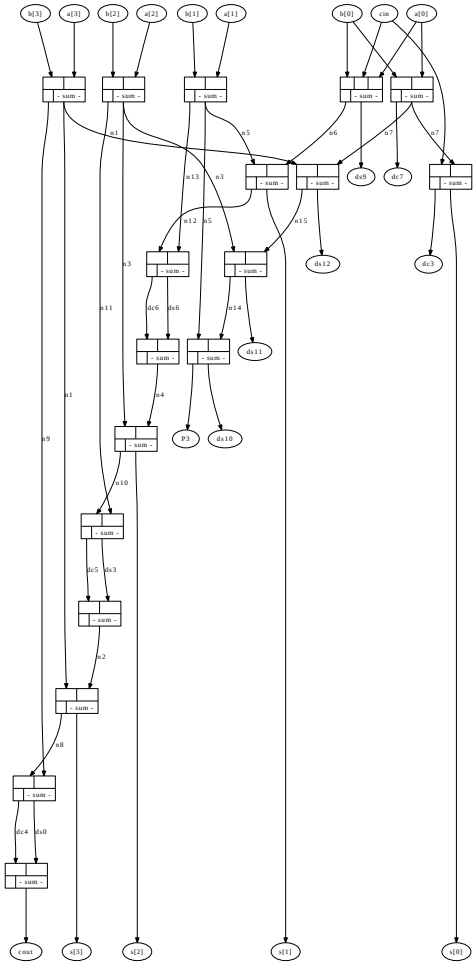


Figure 5.8. Parallel Prefix Adder network.

The output signature is obtained directly from the encoding of the output bits,

$$Sig_O(N_{A4}) = 16c_{out} + 8s_3 + 4s_2 + 2s_1 + s_0$$

Hence the expected reference signature is:

$$[r_O,\ r_I]^T \begin{bmatrix} x_O \\ -x_I \end{bmatrix} = 16c_{out}+8s_3+4s_2+2s_1+s_0-a_o-2a_1-4a_2-8a_3-b_0-2b_1-4b_2-8b_3-c_{in}$$

In order to compute an algebraic signature, first we try to solve the system for α by forcing r_S to zero using the following relationship:

$$A^T\ \alpha = \begin{bmatrix} -r_I \\ r_O \\ 0 \end{bmatrix}$$

In this case the system has no solution. Therefore, we try to complete the signature in terms of r_S. Subsequently we must consider r_S to be free signals and find α using the following relationship:

$$[A_I,\ A_O]^T\ \alpha = \begin{bmatrix} -r_I \\ r_O \end{bmatrix}$$

This system has solution and the corresponding algebraic signature is:

$$\begin{aligned} & r^T\ x = 16c_{out} + 8s_3 + 4s_2 + 2s_1 + s_0 \\ & -a_o - 2a_1 - 4a_2 - 8a_3 - b_0 - 2b_1 - 4b_2 - 8b_3 - c_{in} \\ & -2dc_7 + 2n_6 - 4n_{13} - 8n_{11} - 16n_9 - 2dc_3 + 16dc_4 + 16ds_0 \\ & +8n_2 + 4n_4 - 16n_8 - 8n_{10} - 4n_{12} \end{aligned}$$

In this case the algebraic signature has a non-empty residual expression:

$$\begin{aligned} & RE(N) = r_S^T\ x_S = 2dc_7 - 2n_6 - 4n_{13} + 8n_{11} + 16n_9 + 2dc_3 - 16dc_4 - 16ds_0 \\ & -8n_2 - 4n_4 + 16n_8 + 8n_{10} + 4n_{12} \end{aligned}$$

associated with the internal signals. To prove that the design is correct we must check if the residual expression evaluates to zero for all possible values of x_S produced by the network. This can be done by imposing Boolean constraints as discussed earlier. From the network shown in Figure 5.8, we can see that the variables dc_4, dc_5 and dc_6 are zero (this is the relation that we proved earlier in the OR configuration shown in

Figure 5.2(a)). It can also be seen that the signals ds_9 and s_0 share the same inputs and hence they are equal. This can be modeled by adding the constraints:

$$dc_4 = 0$$
$$dc_5 = 0$$
$$dc_6 = 0$$
$$ds_9 = s_0$$

to the linear system $A\ x = b$. The new system has solution and the corresponding algebraic signature is:

$$r^T\ x = 16c_{out} + 8s_3 + 4s_2 + 2s_1 + s_0$$
$$-a_o - 2a_1 - 4a_2 - 8a_3 - b_0 - 2b_1 - 4b_2 - 8b_3 - c_{in}$$

This signature is identical to the reference signature and it can be seen that there is no residual expression proving that the design is correct.

5.4 Verifying Incorrect Designs

In this section we show the effect of incorrect design on its algebraic signature. The following Lemma is a direct consequence of Theorem 5.1.3

Lemma 5.4.1. *Given an arithmetic circuit N represented by $A\ x = b$ with reference signature $Ref(N) = [-r_I, r_O]^T \begin{bmatrix} x_I \\ \\ x_O \end{bmatrix}$, if $RE(N) = r_S^T\ x_S \neq 0$ i.e., it does not evaluate to zero for all the values of x_S produced by the network, the circuit is incorrect.*

Example: To show the signature of an incorrect design we consider the example of a signed 2×2 multiplier network that will make it incorrect by introducing some

modifications. The design is modified in such a way that the inputs to HA_3 are x_1 and x_6 instead of x_1 and x_5 as it would be in a correct design. The incorrect version of the network, shown in Figure 5.9.

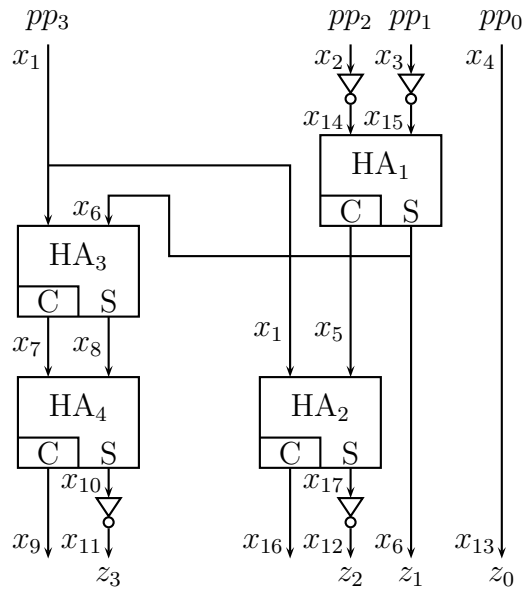


Figure 5.9. Incorrect signed 2×2 multiplier network with signals x_5 and x_6 incorrectly routed.

The network is described by the following equations:

$$\begin{cases} x_{14} + x_{15} - x_6 - 2x_5 = 0 \\ x_1 + x_5 - x_{17} - 2x_{16} = 0 \\ x_7 + x_8 - x_{10} - 2x_9 = 0 \\ x_1 + x_6 - x_8 - 2x_7 = 0 \\ x_2 + x_{14} = 1 \\ x_3 + x_{15} = 1 \\ x_4 - x_{13} = 0 \\ x_{17} + x_{12} = 1 \\ x_{10} + x_{11} = 1 \end{cases} \tag{5.9}$$

or, equivalently in matrix $A\ x = b$ with:

$$\mathrm{A} = \begin{array}{c} \begin{array}{cccc|cccc|cccccccc} x_1 & x_2 & x_3 & x_4 & x_{11} & x_{12} & x_6 & x_{13} & x_5 & x_7 & x_8 & x_9 & x_{10} & x_{14} & x_{15} & x_{16} & x_{17} \end{array} \\ \left(\begin{array}{cccc|cccc|cccccccc} 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -2 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right) \end{array} \quad \mathrm{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

The reference signature for the network remains the same as in the previous case as it refers to a correct (reference) design

$$Ref(N) = [-r_I,\ r_O]^T \begin{bmatrix} x_I \\ \\ x_O \end{bmatrix} = -8x_{11} + 4_{12} + 2x_6 + x_{13} - 4x_1 + 2x_2 + 2x_3 - x_4$$

As before, we try to solve the system for α by forcing r_S to zero. In this case, the system has no solution. So we use the following equation to compute the signature

$$[A_I,\ A_O]^T\ \alpha = \begin{bmatrix} -r_I \\ \\ r_O \end{bmatrix}$$

This system has solution, $\alpha^T = [-6\ \ -4\ 0\ 0\ \ -8\ 4\ \ -1\ 2\ 2\ 0]$ and the corresponding algebraic signature is:

$$\begin{aligned} r^T\ x &= -8x_{11} + 4x_{12} + 2x_6 + x_{13} - 4x_1 + 2x_2 + 2x_3 - x_4 \\ &+12x_5 + 8x_7 + 4x_8 - 8x_{10} - 4x_{14} - 4x_{15} + 4x_{17} \end{aligned}$$

By construction, the input/output signatures match those of the reference signature (otherwise the system will have no solution) but the algebraic signature has a residual

expression, different than before

$$r_s^T \; x_s = -12x_5 - 8x_7 - 4x_8 + 8x_{10} + 4x_{14} + 4x_{15} - 4x_{17}$$

Unlike in the previous example, this expression does not evaluates to zero for all possible values of x_S produced by the network. This is because the signals x_8, x_{17} and x_7, x_{16} are not equivalent which makes $RE(N) \neq 0$. This can also be confirmed by simulation.

$$RE(N) = -12x_5 - 8x_7 - 4x_8 + 8x_{10} + 4x_{14} + 4x_{15} - 4x_{17} \neq 0$$

With the residual expression not equal to zero the algebraic signature is not equal to the provided reference signature, proving that the design is incorrect.

CHAPTER 6

SATISFIABILITY MODULO THEORY (SMT)

6.1 Introduction to SMT

An important direction of research in functional verification looks into Satisfiability Modulo Theories (SMT) solvers. SMT is a generalization of Boolean SAT, in which the binary variables are replaced by predicates or binary-valued functions of non-binary variables. These predicates may come from a variety of underlying theories. SMT solvers combine Boolean SAT with specialized solvers for some well-defined theories, such as Boolean logic, linear arithmetic, theory of equality of uninterrupted functions, and others [4] [19]. Some of the SMT solvers for quantified free linear arithmetic include MathSAT [37], YICES [38], Z3 [39], and for non-linear arithmetic ABSolver [3] and iSAT [36]. New SMT solvers are developed continually, including those to efficiently solve linear inequalities in integer domain (Mistral) [16]. Our work is related to this topic but tackles the problem in a way that does not require solving a decision problem or finding a satisfying assignment for the integer variables.

Satisfiability Modulo Theories (SMT) [2] check if a formula is satisfiable or not when they are given in first-order logic, with associated background theories. If the formula is satisfiable, SMT returns a satisfying solution; otherwise it generates a proof of unsatisfiability. SMTs are used in various applications [2] such as hardware verification at higher levels of abstraction (RTL and above), verification of analog/mixed-signal circuits, verification of hybrid systems, software model checking, software testing and in finding vulnerabilities, verifying electronic voting machines, and in the field of security.

The formulas used in SMT are specified in first-order logic. First-order logic is a mathematical notation with expressions involving propositional symbols, predicates, functions and constant symbols and quantifiers. The difference between first-order logic and propositional (Boolean) logic is that the latter involves only propositional symbols and operators. In contrast, first-order logic are made up of sequences of symbols. Some of the useful theories [2] used in SMT are given below:

- Equality (with uninterpreted functions)
- Linear arithmetic (over Q or Z)
- Difference logic (over Q or Z)
- Finite precision bit-vectors
- Arrays/memories
- Miscellaneous :Non-linear arithmetic, strings, sets, etc

Theory of equality and uniterpreted functions (EUF) is also called as *free theory* since function symbols can take any meaning. Finite-precision bit-vector arithmetic are used to solve arithmetic (add, subtract, multiply, divide) and bit-wise logical (and, or, xor) operations. Theory of linear arithmetic involves a Boolean combination of linear constraints and are used in verification of analog circuits. Difference logic is a Boolean combination of linear constraints and is used in processor datapath verification. Arrays/memories are useful in modeling data structures in both software and hardware [2].

6.2 SMT Solvers

The introduction of Satisfiability Modulo Theory (SMT) resulted in the development of a number of satisfiability solvers. These solvers were developed in both

academia as well as industry. A Satisfiability Modulo Theories Competition (SMT-COMP) [34] is conducted every year to enhance advances in SMT, especially in the field of hardware and software verification. The Satisfiability Modulo Theories Library (SMT-LIB) [35] provides a common description for the background theories and a library of benchmarks. The SMT-LIB also specifies a common input and output format for SMT solvers. A number of SAT solvers are currently available online. In this thesis we compared our approach with three of those solvers which have performed very well in the SMT-COMP in the field of integer arithmetic namely MathSAT, Yices and Z3.

6.2.1 MathSAT

MathSAT [37] is a SMT solver for formal verification. This is a joint project of DISI-University of Trento and FBK-IRST. MathSAT employs a DPLL[14, 13, 21, 23]-based decision procedure for solving the SMT problem for various theories like Equality and Uninterpreted Funtion (EUF), linear arithmetic over the Reals (LA(R)) and over the intergers (LA(Z)) and Difference Logics (DL). MathSAT integrates the state of the art SAT solver along with a variety of solvers for different theories and performs optimization. MathSAT has been used in different applications varying from formal verification of inifinite state systems to equivalence checking and model checking of RTL hardware designs.

6.2.2 Yices

Yices [38] is an efficient and flexible, high performance SMT solver developed by the Computer Science labaratory at SRI International. It supports the theories that are part of the SMT-LIB library. Some of the theories supported by Yices are linear real and integer arithmetic, extensional arrays, fized-size bit-vectors, quanitifers, lambda expressions, etc. Yices is also freely available to users. Yices architecture

integrates a DPLL-based SAT solver along with a core theory solver which handles EUF and satellite theories to support arithmetic, arrays, etc.

6.2.3 Z3

Z3 [39] is a fast and efficient SMT solver, developed at Microsoft Research. Z3 also supports all the theories that are part of the SMT-LIB library. Some of the theories that are supported by Z3 are linear real and integer arithmetic, fized-size bit vectors, quantifiers, extensional arrays and uninterpreted functions. Z3 also has various testing and verification tools from Microsoft Research integrated with it. Like Yices, Z3 also has an architecture which integrates a DPLL-based SAT solver, a core theory solver and a satellite theory solver. Apart from that, Z3 also has an E-matching abstract machine for solving quantifiers. The advantage of Z3 over other solvers is that it integrates different theory solvers in an efficient combination and also allows new theories to be added to the architecture without modifying the core.

6.3 Relation of the Proposed Verification Method to SMT Techniques

In principle, given the system of linear equations $A\ x = b$, describing network N, and the reference signature $Ref(N)$, checking if the network satisfies the reference signature can be modeled as satisfiability (SAT) problem. Specifically, we need to show that:

$$(Ax = b) \wedge (Ref(N) \neq 0) \tag{6.1}$$

is unsatisfiable (unSAT), i.e., we need to show that it will never be the case that the network N does not satisfy $Ref(N)$.

We performed this test on a number of arithmetic circuits with known and proved functionality using two SMT solvers that support linear integer arithmetic, namely Yices [38], and Z3 [39].

6.4 Comparison with SMT solvers

The experiments were conducted on a 2Ghz machine running Linux, with Intel(R) Dual Core(TM) T3200 processor and 3GB RAM. The results of the experiment with SMT solvers are shown in Table 6.1 for unsigned multipliers up to 32 × 32 bits and for a Prefix Adder. The SMT solvers were given the same set of constraints as our solver. As shown in the table, Yices and our method can solve the Prefix Adder very

Design	Z3 (sec)	Yices (sec)	Our method (sec)
$mult\ 3 \times 3$	0.23	0.02	≤ 0
$mult\ 4 \times 4$	466.36	0.05	≤ 0
$mult\ 8 \times 8$	MO	TO	≤ 0
$mult\ 16 \times 16$	MO	TO	0.02
$mult\ 24 \times 24$	MO	TO	0.04
$mult\ 30 \times 30$	MO	TO	0.07
$mult\ 32 \times 32$	MO	TO	0.09
$Prefix\ Adder$	160.31	0.05	0.01

Table 6.1. Comparison with SMT solvers (without Boolean constraints, $RE \neq \phi$). (MO = out of memory 4GB, TO = timeout after 1800sec)

fast, while Z3 needs 160 seconds. (by the way, without the HA constraint expressed by equation $x_C + x_S \leq 1$ that solution is 460 sec). Neither of the SMT solvers were able to solve the problem for multipliers with more than 8 bits, while our method computes the signature (with a non-empty residual expression) in a fraction of a second for multipliers upto 32 bits. Z3 runs out of memory (4GB) while Yices is unable to complete the computation in 30 minutes.

We conducted another set of experiments by adding some of the Boolean constraints discussed in Section 5.2. The results are shown in Table 6.2 for unsigned multipliers up to 32 × 32 bits. The addition of Boolean constraints (OR configuration) resulted in our method computing the signature that is free of residual ex-

pressions. From the table, we can see that our method computes the signature much faster than the SMT solvers.

Design	Z3 (sec)	Yices (sec)	Our method (sec)
$mult\ 3 \times 3$	0.01	0.01	≤ 0
$mult\ 4 \times 4$	0.02	0.02	≤ 0
$mult\ 8 \times 8$	0.07	0.04	≤ 0
$mult\ 16 \times 16$	0.81	0.18	0.02
$mult\ 24 \times 24$	7.00	0.29	0.03
$mult\ 30 \times 30$	21.88	0.54	0.05
$mult\ 32 \times 32$	28.98	0.58	0.06

Table 6.2. Comparison with SMT solvers (with Boolean constraints, $RE = \phi$).

While the application of SMT solvers to property and model checking is unquestionable, their use in functional verification of custom arithmetic circuits remains to be explored. This is because the decision-based verification methods can only check a limited number of properties, explicitly given by the designer, and it is difficult to translate a problem of testing the circuit functionality into a finite number of properties. The size of the resulting decision space is likely to be prohibitive. These solvers have potential to solve the arithmetic verification problem, but need to be enhanced with new, more efficient and adequate bit-level arithmetic models. We believe that the algebraic formulation of the arithmetic verification problem presented in this work can be used to enhance capabilities of SMT solvers by introducing a new model based on computing the algebraic signature for a circuit.

CHAPTER 7

ANALYSIS OF RESULTS

7.1 Functional Verification System

A detailed flow of the arithmetic verification procedure based on algebraic signature computation is shown in Fig. 7.1. The input to the system is the description of the arithmetic network N, composed of arbitrary logic gates, HA and FA operators, along with the reference signature provided by the designer. The system computes a complete signature of the network and reports if there is a non-empty residual expression $RE(N)$ that needs to be examined.

If the residual expression contains internal signal variables, additional constraints may need to be learned from the network, as described earlier, and imposed on the system. The system is solved again with an enhanced system of constraints $A'x' = b'$.

Note that by construction (equation 4.4) the signature vector of a correctly designed circuit will always match its reference signature, otherwise the system has no solution, and the circuit is declared incorrect.

7.2 Experimental Setup

The algebraic verification technique described here has been implemented as a prototype program written in C. The program uses GNU Linear Programming Kit GLPK package [44] to solve the associated linear system and follows the flow shown in Fig. 7.1. Using this program, we conducted a set of experiments on a number of arithmetic circuits, including different adders and large integer multipliers. The input

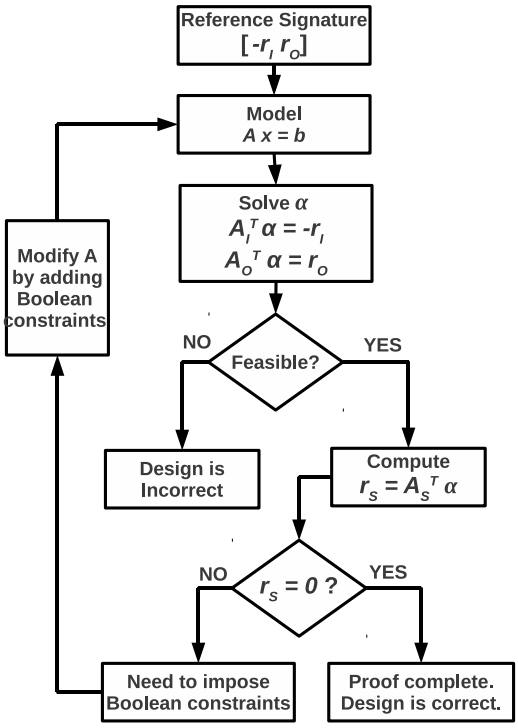


Figure 7.1. Flowchart of the functional verification system.

to the program is the expected reference signature and the output is the complete algebraic signature in terms of all the variables in the network.

The experiments were conducted on a 2Ghz machine running Linux, with Intel(R) Dual Core(TM) T3200 processor and 3GB RAM. For multiplier designs, we used a multiplier generator software, courtesy of the University of Kaiserslauten [42], to generate a bit-level structural verilog code. For different adder implementations we wrote RTL codes for different adder implementations in Verilog. The RTL code was then synthesized using Synopsys Design Compiler [15]. We used the Ohio State University's Standard Cell library [24], which is available online, for synthesizing our adders in structured form (composed of only full-adders) or gate-level (composed of logic gates). The verilog code was parsed using our home-grown parser to produce a network of HA, FA and basic logic gates. A system of linear equations was generated

from the netlist, as described in Section 4.1. Finally, a program written by André Rossi [28] with link to GLPK was used to generate the signature for the network, given the expected reference signature.

7.3 Gate-level Arithmetic Circuits

First, we examined the case when the arithmetic circuit is represented as gate-level network. We carried out a set of experiments with different types of adders. Some of the different adders we examined were Carry Look-Ahead Adder, Ripple Carry Adder and Parallel Prefix Adder. Equations 3.9 and 3.12 show how to derive the input and output signatures for binary adders, using primary inputs and primary outputs.

7.3.1 Carry Look-Ahead Adder

The carry look-ahead adder was implemented in RTL and then synthesized using Synopsys Design Compiler with a standard cell library from Ohio State University. The synthesized adders were strictly gate-level designs. The number of gates used in the designs were approximately equal to twice the number of constraints reported in Table 7.1. We carried out a set of experiments for adders upto 256 bits. The table gives the following data: the size of the adder (in the number of bits n of each operand); the number of linear equations (*constr*); and the CPU time of our method to compute the signature.

It should be noted that the results tabulated above gives a solution with empty residual expression. However, this was obtained only after adding some additional constraints to the network. Without adding the additional constraints we got a solution with non-empty residual expression. Once the internal variables of $RE(N)$ were identified, it was an easy task to prove that these signals are always zero in the context of the circuit. The carry look-ahead adder network had the OR configuration

	CarryLookAheadAdder	
$Size(n)$	*Constr.*	$CPU(sec)$
4	20	≤ 0
8	40	≤ 0
16	80	≤ 0
24	120	≤ 0
32	160	0.010
64	320	0.010
128	640	0.020
256	1024	0.030

Table 7.1. CPU runtime for computing algebraic signature for n-bit carry look ahead adder with basic logic gates ($RE = \phi$).

and by adding the constraint that the internal signal C of the OR configuration, whose inputs come from two half-adders (reconvergent fanout), is equal to zero, we obtained a solution with empty residual expression. We had already proved that the internal signal C in the OR configuration is zero in Section 5.3.

	RippleCarryAdder	
$Size(n)$	*Constr.*	$CPU(sec)$
8	8	≤ 0
16	16	≤ 0
24	24	≤ 0
32	32	≤ 0
48	48	≤ 0
64	64	0.010
128	128	0.010

Table 7.2. CPU runtime for computing algebraic signature for n-bit ripple carry adder in structured form (composed of full adders) ($RE = \phi$).

7.3.2 Ripple Carry Adder

The ripple carry adder, like carry look-ahead adder, was implemented in RTL and then synthesized using Synopsys Design Compiler with a standard cell library from Ohio State University. We synthesized these adders in two forms: the designs were

represented in structured form (composed of full-adders) and in the other form the adders were represented as gate-level designs. The number of gates used in the designs were approximately equal to twice the number of constraints reported in Table 7.3. We carried out a set of experiments for adders upto 256 bits. The table gives the following data: the size of the adder (in the number of bits n of each operand); the number of linear equations (*constr*); and the CPU time of our method to compute the signature.

It should be noted that the results tabulated in Table 7.2 gives a solution with empty residual expression. This is quite obvious considering the fact these designs were in structured form (composed of only full-adders). We have already seen in Section 4.2 that when the designs are represented with only full-adders, we get a solution with empty residual expression. Hence there was no necessity to add any additional constraints.

	$RippleCarryAdder$	
$Size(n)$	$Constr.$	$CPU(sec)$
4	32	≤ 0
8	64	≤ 0
16	64	≤ 0
24	192	≤ 0
32	256	≤ 0
64	256	0.010
128	512	0.020
256	1024	0.030

Table 7.3. CPU runtime for computing algebraic signature for n-bit ripple carry adder with gate level implementation ($RE \neq \phi$).

The results shown in Table 7.3 gives a solution with non-empty residual expression. For these class of designs, we need to examine additional constraints that need to be added to the network to obtain a solution with empty residual expression. By analyzing the ripple carry adder network we found out that these designs did not

have the OR configuration as in the case of carry look-ahead adders or any of those Boolean constraints we discussed in Section 5.2. Hence we need to further investigate to find out the additional constraints that might be needed to obtain an empty residual expression. This is an open problem and is considered to be a future work.

7.3.3 Parallel Prefix Adder

We used an example of a Parallel Prefix Adder, taken from [52] and shown in Figure 5.7. We implemented this design in Verilog and then synthesized as earlier using the Ohio State library. Initially we solved the original problem $A\ x = b$ with the equality constraints obtained from the network structure, without any additional constraints. We obtained a non-empty residual expression in terms of the internal signals. It was found that the combination of all the internal signals evaluated to 0. We proved this by simulation. Later by examining the network structure and by adding some of the Boolean constraints discussed earlier, we obtained an empty residual expression. Thus proving that the computed algebraic signature matches the reference signature. But for bigger designs which had multiple fan-outs, we were not able to identify the Boolean constraints that might be necessary to obtain an empty residual expression. Hence as in the case of ripple carry adders, we need to further investigate to find out the additional constraints that might be needed to obtain an empty residual expression. This is an open problem and is considered to be a future work.

7.4 Multipliers

We also experimented with different multipliers, including Booth-encoded, and non-Booth multipliers. Since multipliers are non-linear networks, we first needed to transform their description into a linear form suitable for our system. This can be done readily by partitioning the design into two blocks: Boolean circuit G, which

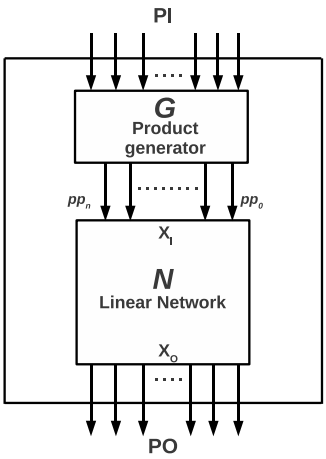


Figure 7.2. Deriving inputs for multiplier networks.

generates partial products (using known recoding scheme) and a linear network N which uses these partial/recoded products as inputs, as shown in Figure 7.2.

Circuit G can be verified trivially using Boolean methods, such as ABC [22], as long as the recoding scheme used to design the circuit is given. The input to our linear system can then be easily extracted from the boundary of the two blocks.

Equation 4.7 shows how to derive the reference signature for area multiplier, using partial product, $a_i b_j$. Similar expressions can be readily obtained for Booth-recoded products. In fact, we have derived expressions for such input signatures for radix 2 Booth multipliers using the known Booth recoding methods and used them to prove Booth multipliers, obtaining CPU times similar to those shown in Table 7.4.

Table 7.4 shows our results for a set of integer unsigned multipliers up to 256 bits. The table gives the following data: the size of the multiplier (in the number of bits n of each operand); the number of linear equations (*constr*); the CPU time of our method to compute the signature for unsigned multipliers.

Size (n)	Multipliers $Unsigned(n \times n)$	
	$Constr.$	$CPU(sec)$
3	21	≤ 0
4	44	≤ 0
8	216	≤ 0
16	944	0.020
24	2184	0.040
30	3450	0.070
32	3936	0.110
53	10971	0.760
64	16064	1.570
128	64896	25.450
192	146496	136.290
256	260864	446.950

Table 7.4. CPU runtime for computing algebraic signature for n-bit integer multipliers without additional constraints ($RE \neq \phi$).

It should be noted that the results shown in Table 7.4 gives a solution with non-empty residual expression. However, these expressions had an interesting property which made it very easy to prove that $RE(N) = r_S^T\ x_S = 0$. Namely, all coefficients r_S of this expression had the *same sign*. This reduces the proof $r_S^T\ x_S = 0$ to showing that each signal x_S in the expression is *independently* zero. Once the internal variables of $RE(N)$ are identified, it was an easy task to prove that these signals are always zero in the context of the circuit. Using the example of the circuit in Figure 4.1, it is easy to prove that $x_9 = 0$. Similar cases were found in the multiplier circuits. In the case of the multipliers in Table 7.4 the only signals x_S in $RE(N)$ are the internal signals C of the OR configuration, whose inputs come from two half-adders (reconvergent fanout). The experiments in Table 7.4 were repeated with the Boolean constraints imposed on the network, as discussed earlier in Section 5.2 and the signatures were computed. The computed signatures were free of residual expressions and the results are shown in Table 7.5. The CPU time for arithmetic proof (AP) of integer multipliers, reported

$Size(n)$	$Multipliers$				AP [18] (sec)
	$Unsigned(n \times n)$		$Signed(n \times n)$		
	$Constr.$	$CPU(sec)$	$Constr.$	$CPU(sec)$	
4	36	≤ 0	44	≤ 0	-
8	264	≤ 0	184	≤ 0	-
16	720	0.020	752	0.010	-
24	1656	0.030	1704	0.040	7
32	2976	0.060	3040	0.060	-
53	8268	0.320	8374	0.330	480
64	12096	0.630	12224	0.640	840
128	48768	8.750	49024	8.750	-
192	110016	45.230	110400	46.210	-
256	195840	151.950	196352	153.930	-

Table 7.5. CPU runtime for computing algebraic signature for n-bit integer multipliers with additional constraints propagated in the network ($RE = \phi$).

in [18] is also shown. The AP results were computed on a comparable 64-bit 2GHz Power5 machine, and reported only for 24, 53 and 64 bit integer multipliers. No larger multipliers were reported for the purpose of comparison.

$Size(n)$	$Booth - encoded Multipliers$			
	$Unsigned(n \times n)$		$Signed(n \times n)$	
	$Constr.$	$CPU(sec)$	$Constr.$	$CPU(sec)$
4	46	≤ 0	34	≤ 0
8	148	≤ 0	118	≤ 0
16	496	0.040	430	0.040
24	1036	0.150	934	0.130
32	1768	0.410	1630	0.360
64	6616	6.210	6334	5.940
128	25528	126.900	24958	134.770

Table 7.6. CPU runtime for computing algebraic signature for n-bit integer Booth-encoded multipliers with additional constraints propagated in the network ($RE = \phi$).

Table 7.6 shows our results for a set of integer Booth-encoded signed and unsigned multipliers. The columns are organized similarly to those in Table 7.5. The results reflect computing signature with no residual expression.

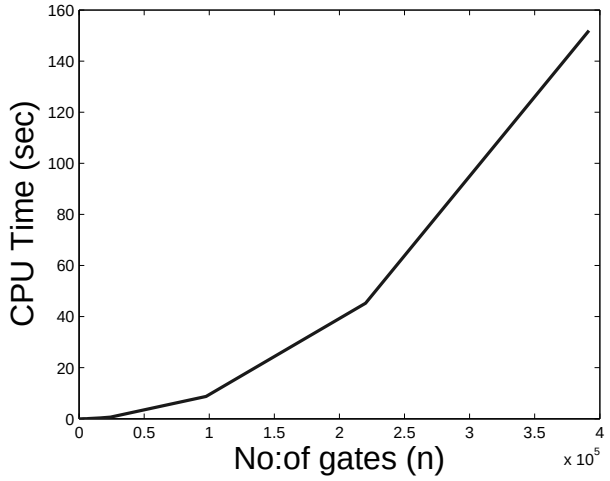


Figure 7.3. Computational complexity of our approach.

Figure 7.3 shows the computational complexity of our approach where CPU runtime is plotted against the number of gates in the design. The number of gates used in the design is approximately equal to twice the number of constraints reported in Table 7.5. It can be observed from Figure 7.3 that, in contrast to ILP methods, whose runtime complexity are exponential, the computational complexity of our approach is polynomial. The graph shows that the complexity is below $O(N^2)$. This is predictable since we never solve a decision problem or an ILP problem in integer domain. This confirms our claim that our method is efficient and scalable.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

We developed a new methodology and a prototype system for functional verification of arithmetic circuits based on computing algebraic signature. Such a signature uniquely describes the behavior of the design, and as such it can be useful for different forms of functional verification. The purpose of this work was to show a potential of the proposed technique to verify functionality of arithmetic networks. For arithmetic designs, such as multipliers, the completion of the signature was followed up by a proof that the residual signature expression reduces to zero. This issue has been investigated, the complexity of the problem seems to be significantly smaller than the original one. This was done as a part of this work for certain class of designs. This is one of the main thrusts of our work.

The proposed technique can be used in Functional verification, by providing the reference signature. In this case, the system will complete the signature in terms of the internal variables. If the system is feasible, the reference signature is consistent with the network, which proves its functionality specified by the signature. There are two important cases to be considered here: a) $r_S = 0$ where the internal signals are eliminated from the input/output relationship, which was also the case in the 7-3 counter design; and b) $r_S \neq 0$. It can be shown that if the reference signature is correct, the residual polynomial $r_S^T \; x_S$ reduces to zero for all values of variables in x_S produced by the network. If the reference signature is incorrect, the residual expression does not reduce to zero.

The following are the possible directions in which this work can be carried forward in the future:

• RTL and gate-level Verification

The proposed technique can be also used to verify RTL and gate-level networks against their behavioural arithmetic specifications. This can be implemented by developing techniques and algorithms to verify both RTL and gate-level networks against their behavioral arithmetic specifications. For gate-level implementations, the gate-level network can be converted to a network of half-adders, using the proposed modeling of gates as HAs and FAs. The model can be enhanced with additional constraints ($x_c + x_s \leq 1$) properly turned into equalities using slack variables. Additional Boolean invariants may need to be learned and derived from the circuit, as discussed earlier, in order to obtain a signature without a residual expression. If the resulting residual expression contains internal signals, the content of that expression needs to be examined and proved to be zero using Boolean methods.

Our technique can be readily extended to arithmetic circuits specified at RTL and higher levels. The basic algebraic equations for HA and FA, shown in Section 3.1, can be extended to RTL arithmetic descriptions with word-level arithmetic operators. For example, an n-bit adder $A + B$ can be written as a single equation

$$A + B = 2^n C + S$$

Where variables A, B and S are n-bit words and C is a carry-out bit. Additional encoding information in n bits can be provided as additional equation, $\sum_{i=0}^{n-1} 2^i \ S_i$ if needed.

• Property Checking

The described technique is directly applicable to property checking. This can be accomplished by representing the property P by its own algebraic signature $Sig_P(N) = r^T \ x_P$ and checking if it is consistent with the reference signature of the network

using equation $A_P^T\ \alpha = r_P$, where index P refers to the "property" to be proven. The feasibility of the resulting linear system will indicate whether such a consistency is maintained, and hence if the property is satisfied.

• Equivalence Checking

The system can also be used for equivalence checking between different levels of design description, by computing and comparing the signatures of the two designs under comparison.

• Extracting Circuit Behavior

The proposed signature-based method will also be used to *extract functional behavior* of the design. This can be done by using only the output signature $Sig_O(N) = r_O^T\ x_O$ to compute the input signature, $Sig_I(N) = r_I^T\ x_I$. Given the output signature vector r_O, we first solve the equation, $A_O^T\ \alpha = r_O$ and use the computed vector α to obtain input signature vector as $r_I = -A_I^T\ \alpha$. The result can be trusted only if the residual expression RE is zero. This is the case of the 7-3 counter in Fig. 3.3, equation (3.6). Extraction of a design behavior is a unique feature, which to the best of our knowledge, is not offered by any other system.

• Debugging

Investigation on how to use the residual expression to *identify* bugs in the circuit can be done. This can be done by analyzing the internal signal variables if the Simplex phase I (presolver) of the LP solver fails. If LP is infeasible, the non-zero values assumed by these variables provide essential information which equations (i.e., which arithmetic operators) prevent the LP from being feasible. In addition, if LP is feasible but the residual expression RE cannot be reduced to zero, the content of the resulting RE will be used to reason about bugs. This would be an ambitious but a novel and worthwhile task.

• Datapath Verification

The proposed verification method can be extended to handle datapaths. The main

challenge on this front would be to obtain a reference signature for the entire datapath. Such a signature may not be available in a closed analytical form, required by our system, or can be hidden in the HDL description of the design. In this case the design will have to be partitioned into smaller blocks for which reference signatures can be generated with the help of Boolean solvers, such as ABC [22].

APPENDIX

FUNCTIONAL VERIFICATION FLOW

Multiplier Generator:

The different multiplier designs used in this thesis were generated by a multiplier generator software obtained from the University of Kaiserslauten [42] which produced a bit-level structural verilog code. We experimented with different multipliers, including Booth-encoded, and non-Booth multipliers. The software gives the description of the generated circuit in Blif, Verilog and VHDL. The partial products description of the generated circuit is presented in Blif. A brief description of the options available in the software is given below:

./genmult	Options
-a $\langle int \rangle$	Bit-width of multiplicand
-b $\langle int \rangle$	Bit-width of multiplier
-s	Signed multiplication Default - Unsigned
-m	Signed non-Booth Baugh-Wooley scheme
	Default - Modified non-Booth Baugh-Wooley scheme
-e $\langle 1 \vert 2 \rangle$	Default - non-Booth encoded algorithm is applied
	Type 1 - Booth encoded algorithm, using logic units
	Type 2 - Booth encoded algorithm, using arithmetic units
	ATTENTION: $\langle -e2 \rangle$ is compatible with following options only:
	-a, -b, -t, -o (requires -v 1), -v 1, -f, -g, -r
-t	Tree structure of adders
	Default - Cascade structure
-o $\langle filename \rangle$	Create circuit file
-p $\langle filename \rangle$	Create part-prod Blif file
-f $\langle filename \rangle$	Create part-prod scheme file
-g $\langle filename \rangle$	Create test bench file (a theorem for 'GateProp'-tool)
	(option -v is required to be next)

-c	Generate compile-tauri file for 'GateProp'-tool and/or
	specify there the circuit file to be analyzed by vhdl2gates,
	print on screen: name of circuit file and main entity
	(optionally specify after corresponding -o and -v ⟨2⟩)
-+	Separate command line data between two multipliers
-N ⟨*int*⟩	1. Number of all specified multipliers
	specify after all multipliers related data only !!!
	2. Separate command line data between multipliers and adder
	default by N = 1
-v ⟨1|2⟩	The format to print a complete circuit or a partial product part
	Default - Blif format
	Type 1 - Verilog format (the next after option -o or -g)
	Type 2 - VHDL format (the next after option -o or -g)
-r	Read signal names meaning
Remark:	1. (-o, -p, -f, -g) ⟨*cout*⟩ - print data on the screen
	2. The count of specified circuits begins from zero

Table A.1. Options available in the multiplier generator.

We illustrate the use of the software on a signed 2×2 multiplier. It can be generated by the following command:

./genmult -a 2 -b 2 -s -o mults2b.v -v1

The following verilog file is generated by the software:

```
// generated by:  ./genmult -a 2 -b 2 -s -o mults2b.v -v1
// signed 2x2 standard non-Booth multiplier (modif. Baugh-
Wooley) with primary inputs: a, b primary output: p
module mult_sn_2x2(p, a, b);
// The input-vectors of the circuit #0:
input [1:0] a;
input [1:0] b;
// The only output-vector:
output [3:0] p;
// *****  Partial products *****
assign pp0 = a[0] &b[0];
assign pp1 = a[1] &b[0];
assign nn0 = ~pp1;
assign pp2 = a[0] &b[1];
assign nn1 = ~pp2;
assign pp3 = a[1] &b[1];
// *****  Partial products *****
// *****  Adders *****
ha sum_sa0_0 (sa0, nn1, nn0, ca0);
assign sa1 = ca0 ~^ pp3;
```

```
assign ca1 = ca0 | pp3;
assign sa2 = ~ca01;
// carry signal: ca0_2 isn't involved in the process
// *****  Adders *****
// ***   outputs   ***
assign p[0] = pp0;
assign p[1] = sa0;
assign p[2] = sa1;
assign p[3] = sa2;
endmodule

module ha (sum, inp0, inp1, carry);
input inp0, inp1;
output sum, carry;
assign sum = inp0 ^ inp1;
assign carry = inp0 &inp1;
endmodule
```

Equation Format:

The verilog file is converted into a set of equations in the form of half-adders, full-adders and basic logic gates. The conversion of the verilog file to the linear equation format was done by a parser written in *Perl*. The following format is used for the equations:

A half-adder (HA) with inputs a, b and outputs S, C is written as

$$\text{a + b - S - 2*C} = 0$$

A full-adder (FA) with inputs a, b and cin and outputs S, C is written as

$$\text{a + b + cin - S - 2*C} = 0$$

A Buffer with input a and output b is written as

$$\text{a - b} = 0$$

An Inverter with input a and output b is written as

$$\text{a + b} = 1$$

An OR gate with input a, b and output d is written as

$$a + b - S - 2*C = 0$$

$$S + C - d = 0$$

The following is the equation file generated by our parser for the signed 2×2 multiplier shown above.

```
nn0 + pp2 = 1
nn1 + pp1 = 1
nn1 + nn0 - sa0 - 2 * ca0 = 0
ca0 + pp3 + sa1 - 2 * C0 = 1
ca0 + pp3 - S_XR - 2 * C_AND = 0
S_XR + C_AND - ca1 = 0
sa2 + ca1 = 1
p[0] - pp0 = 0
p[1] - sa0 = 0
p[2] - sa1 = 0
p[3] - sa2 = 0
```

Linear Programming Solver:

The Linear Programming (LP) solver used in this thesis was **GNU Linear Programming Kit (GLPK)**[44]. GLPK is a free software available for solving large scale linear programming (LP), mixed integer programming (MIP), and other mathematical programming problems. The package includes several components such as primal and dual-simplex methods, primal-dual interior-point method, branch and cut method, etc. The GLPK package is a set of routines which are organized in the form of a callable library. The program to call the GLPK package, called archi-sig, was written by André Rossi [28]. We developed another parser which takes the equation file as its input and generates an output file which follows the format required by the GLPK solver. The format of the program is as follows:

The first line has 6 integers:

$n\ npi\ nfs\ npo\ nps\ m$

n: Total number of variables (i.e. signals) in the problem,

npi: Number of primary inputs,

nfs: Number of free internal signals,

npo: Number of primary outputs,

nps: Number of signals in the (given) partial signature,

m: Number of constraints in the architecture.

PI // Set of npi numbers in $1, \cdots, n$: indexes of primary inputs

FS // Set of nfs numbers in $1, \cdots, n$: indexes of free internal signals

PO // Set of npo numbers in $1, \cdots, n$: indexes of primary outputs

PS // Set of nps numbers in $1, \cdots, n$: indexes of the signals in the (given) partial signature

vps // Set of nps real multipliers for the partial signature.

The next m lines are built as follows:

$N\ var1\ coeff1\ var2\ coeff2 \cdots\ varN\ coeffN = const$

N: Number of variables in the constraint

$var1$: Index of the first variable,

$coeff1$: Coefficient of the first variable

...

$const$: Constant.

$name$ // [optional] line with n names, separated with a space.

The following is the input file to the GLPK solver for a signed 2×2 multiplier.

```
18 4 10 4 8 11
2 4 7 15
1 3 5 6 8 9 10 11 12 13
14 16 17 18
2 4 7 15 14 16 17 18
2 2 -4 -1 1 2 4 -8
2 1 1 2 1 = 1
2 3 1 4 1 = 1
4 3 1 1 1 5 -1 6 -2 = 0
4 6 1 7 1 8 1 9 -2 = 1
4 6 1 7 1 10 -1 11 -2 = 0
3 10 1 11 1 12 -1 = 0
2 13 1 12 1 = 1
2 14 1 15 -1 = 0
2 16 1 5 -1 = 0
2 17 1 8 -1 = 0
2 18 1 13 -1 = 0
x14
x2
x15
```

```
x3
x6
x5
x1
x17
x16
x8
x7
x9
x10
x13
x4
x6
x12
x11
```

The above file is given as input to the GLPK solver. The solver solves the system using the approach discussed in earlier chapters. It generates an output file which contains the α values for the network and gives a residual expression if the reference and algebraic signatures do not match. The following output file was generated by the solver for the signed 2×2 multiplier.

```
#########################################################################
#                                                                       #
# SIGNATURE CHECKING AND COMPLETION, + RESIDUAL EXPRESSION              #
# Universite de Bretagne-Sud, Lab-STICC                                 #
# Andre Rossi, October 2010.                                            #
#                                                                       #
# This program must be invoked as follows:                              #
# ./archi-sig instance.dat                                              #
#                                                                       #
# Read file_format-sig.txt for more.                                    #
#########################################################################
Reading instance file "mults2b-archi.in"
The layout in "mults2b-archi.in" contains n=18 signals,
  npi=4 of them are primary inputs
  nfs=10 of them are free internal signals
  npo=4 of them are primary outputs
  nps=8 of them are in the (given) partial signature.
There are m=11 constraints in that architecture.
This is ZS (0 elements):
This is PI (4 elements): 2 4 7 15
This is FS (10 elements): 1 3 5 6 8 9 10 11 12 13
This is PO (4 elements): 14 16 17 18
This is PS (8 elements): 2 4 7 15 14 16 17 18
These are the 0 elements of CS (signals whose coefficient
should be found)
CS =
This is PI, the set of primary inputs: 2 4 7 15
This is FS, the set of free internal signals: 1 3 5 6 8 9
```

```
10 11 12 13
This is PO, the set of primary outputs: 14 16 17 18
This is PS, the set of signals whole coefficients in the
signature is known: 2 4 7 15 14 16 17 18
This is vps, the associated coefficients: 2 2 -4 -1 1 2 4 -8
name[1]=z14
name[2]=x2
name[3]=x15
name[4]=x3
name[5]=x6
name[6]=x5
name[7]=x1
name[8]=x17
name[9]=x16
name[10]=x8
name[11]=x7
name[12]=x9
name[13]=x10
name[14]=x13
name[15]=x4
name[16]=x6
name[17]=x12
name[18]=x11
ia_aux[1]=1   ja_aux[1]=1   ar_aux[1]=1
ia_aux[2]=2   ja_aux[2]=2   ar_aux[2]=1
ia_aux[3]=2   ja_aux[3]=3   ar_aux[3]=1
ia_aux[4]=1   ja_aux[4]=3   ar_aux[4]=1
ia_aux[5]=3   ja_aux[5]=3   ar_aux[5]=-1
ia_aux[6]=4   ja_aux[6]=3   ar_aux[6]=-2
ia_aux[7]=4   ja_aux[7]=4   ar_aux[7]=1
ia_aux[8]=5   ja_aux[8]=4   ar_aux[8]=1
ia_aux[9]=6   ja_aux[9]=4   ar_aux[9]=-2
ia_aux[10]=4   ja_aux[10]=5   ar_aux[10]=1
ia_aux[11]=7   ja_aux[11]=5   ar_aux[11]=-1
ia_aux[12]=8   ja_aux[12]=5   ar_aux[12]=-2
ia_aux[13]=7   ja_aux[13]=6   ar_aux[13]=1
ia_aux[14]=8   ja_aux[14]=6   ar_aux[14]=1
ia_aux[15]=9   ja_aux[15]=6   ar_aux[15]=-1
ia_aux[16]=10   ja_aux[16]=7   ar_aux[16]=1
ia_aux[17]=9   ja_aux[17]=7   ar_aux[17]=1
ia_aux[18]=3   ja_aux[18]=9   ar_aux[18]=-1
ia_aux[19]=5   ja_aux[19]=10   ar_aux[19]=-1
ia_aux[20]=10   ja_aux[20]=11   ar_aux[20]=-1
 p=9, p_aux=20, sum = 29

alpha, sol of the LP.
Constraint coef. 1 = 2
Constraint coef. 2 = 2
Constraint coef. 3 = 0
Constraint coef. 4 = -4
Constraint coef. 5 = 0
Constraint coef. 6 = 0
Constraint coef. 7 = 0
Constraint coef. 8 = 1
```

```
Constraint coef. 9 = 2
Constraint coef. 10 = 4
Constraint coef. 11 = -8
This is vcs[], the coefficients associated with CS

This is vfs[], the coefficients associated with free
internal signals
vfs[1] = 2
vfs[3] = 2
vfs[5] = -2
vfs[6] = -4
vfs[8] = -8
vfs[9] = 8
vfs[10] = 0
vfs[11] = 0
vfs[12] = 0
vfs[13] = 8
 f=0.000000
Full signature:
 +2x2 +2x3 -4x1 -x4 +x13 +2x6 +4x3 -8x11 = -8x7 -4x8 +8x10 -4x17

Residual expression:  -8x7 -4x8 +8x10 -4x17
Total CPU time: 0.000 seconds
```

BIBLIOGRAPHY

[1] Alizadeh, B., and Fujita, M. Modular Datapath Optimization and Verification Based on Modular-HED. In *IEEE Trans. on Computer-Aided Design* (September 2010), pp. 1422–1435.

[2] Barrett, C., and Seshia, S. A. Introduction to Satisfiability Modulo Theories (SMT). http://www.eecs.berkeley.edu/ sseshia/.

[3] Bauer, A., Pister, M., and Taqutschnig, M. Tool support for the analysis of Hybrid Systems and Models. In *Proc. Design Automation and Test in Europe* (2007), pp. 1–6.

[4] Biere, A., Heule, M., Maaren, H. V., and Walsch, T. *Satisfiability Modulo Theories in Handbook of Satisfiability.* IOS Press, 2008. Chapter 12.

[5] Brayton, R., and Mishchenko, A. ABC: An academic industrial-strength verification tool. In *Proc. Intl. Conf. on Computer-Aided Verification* (2010), pp. 24–40.

[6] Bryant, R. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers 35*, 8 (August 1986), 677–691.

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

[8] Burch, J. R., Clarke, E. M., McMillan, K. L., and Dill, D. L. Sequential Circuit Verification using symbolic Model Checking. In *Proc. Intl. Design Automation Conference* (1990), pp. 46–51.

[9] Chen, Y., and Bryant, R. E. *PHDD: An Efficient Graph Representation for Floating Point Circuit Verification. In *Proc. Intl. Conf. on Computer-Aided Design* (1997), pp. 2–7.

[10] Cheng, Kwang-Ting, and Krishnakumar, A. S. Automatic generation of Functional vectors using the extended finite state Machine Model. In *Proc. Intl. Conf. on Design Automation of Electronic Systems* (January 1996), pp. 57–79.

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

[12] Clarke, E. M., Fujita, M., and Zhao, X. Hybrid Decision Diagrams overcoming the limitations of MTBDDs and BMDs. In *Proc. Intl. Conf. on Computer-Aided Design* (1995), pp. 159–163.

[13] Davis, M., Logemann, G., and Loveland, D. A Machine Program for the Theorem Proving. *Communications of the ACM 5* (1962), 394–397.

[14] Davis, M., and Putnam, H. A Computing Procedure for Quantification Theory. *Journal of the ACM 7* (1960), 201–215.

[15] Synopsys DC Compiler. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/default.aspx.

[16] Dillig, I., Dillig, T., and Aiken, A. Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In *Proc. Intl. Conf. on Computer-Aided Verification* (July 2009), pp. 233–247.

[17] Kozen, D. Kleene Algebra with tests. In *ACM Transcations on Programming Languages and Systems* (1997), pp. 427–443.

[18] Krautz, U., Wedler, M., Kunz, W., Weber, K., Jacobi, C., and Pflanz, M. Verifying Full-Custom Multipliers by Boolean Equivalence Checking and an Arithmetic BitLevel Proof. In *Proc. Asia and South Pacific Design Automation Conference* (2008), pp. 398–403.

[19] Kroening, D., and Strichman, O. *Decision Procedures, An Algorithmic Point of View*. Springer, 2008.

[20] Marques, J. P., and Sakallah, K. A. GRASP- a search algorithm for propositional satisfiability. In *IEEE Trans. on Computers* (May 1999), pp. 506–521.

[21] Marques-Silva, J., and Sakallah, K. A. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD'96* (1996), pp. 220–227.

[22] Mischenko, A., Chatterjee, S., Brayton, R., and Een, N. Improvements to Combinational Equivalence Checking. In *Proc. Intl. Conf. on Computer-Aided Design* (2006), pp. 836–843.

[23] Moskewicz, M., Madigan, C., Y. Zhao, L. Zhang, and Malik, S. Chaff: Engineering an Efficient SAT Solver. In *Proc. of 38th Design Automation Conf.* (June 2001), pp. 530–535.

[24] OSU Standard Cell Library. http://vlsiarch.ecen.okstate.edu/flows/ OSUFreePDK45/.

[25] Pavlenko, E., Wedler, M., Stoffel, D., and Kunz, W. STABLE: A new QF-BV SMT Solver for hard Verification Problems combining Boolean Reasoning with Computer Algebra. In *Proc. Design Automation and Test in Europe* (2011).

[26] Peymandoust, A., and DeMicheli, G. Application of Symbolic Computer Algebra in High-Level Data-Flow Synthesis. In *IEEE Trans. on Computer-Aided Design* (2003), vol. 22, pp. 1154–1165.

[27] Raudvere, T., Singh, A. K., Sander, I., and Jantsch, A. System Level Verification of Digital Signal Processing application based on the Polynomial Abstraction Technique. In *Proc. Intl. Conf. on Computer-Aided Design* (2005), pp. 285–290.

[28] Rossi, André. Program with link to GLPK. http://www-labsticc.univ-ubs.fr/ rossi/.

[29] Sarbishei, O., Tabandeh, M., Alizadeh, B., and Fujita, M. A Formal Approach for Debugging Arithmetic Circuits. In *IEEE Trans. on Computer-Aided Design* (May 2009), vol. 28, pp. 742–754.

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

[31] Shekhar, N., Kalla, P., Enescu, F., and Gopalakrishnan, S. Equivalence Verification of Polynomial Data-Paths with Fixed-Size Bit-Vectors using Finite Ring Algebra. In *Proc. Intl. Conf. on Computer-Aided Design* (2005), pp. 291–296.

[32] Smith, J., and DeMicheli, G. Polynomial Methods for Compontent Matching and Verification. In *Proc. Intl. Conf. on Computer-Aided Design* (1998).

[33] Smith, J., and DeMicheli, G. Polynomial Methods for Allocating Complex Compontents. In *Proc. Design Automation and Test in Europe* (1999).

[34] Satisfiability Modulo Theories Competition (SMT-COMP). http://www.smtcomp.org/.

[35] Satisfiability Modulo Theories Library (SMT-LIB). http://www.smtlib.org/.

[36] HySAT: A Bounded Model Checker for Hybrid Systems. http://hysat.informatik.uni-oldenburg.de/.

[37] MathSAT 4. http://mathsat4.disi.unitn.it/index.html.

[38] Yices: An SMT Solver. http://yices.csl.sri.com/index.shtml.

[39] Z3: An Efficient SMT Solver. http://research.microsoft.com/en-us/um/redmond/projects/z3/index.html.

[40] Maple. http://www.maplesoft.com.

[41] Mathematica. http://www.wri.com.

[42] Program to generate different types of multipliers.

[43] The MathWorks. http://www.mathworks.com.

[44] GNU, GLPK Linear Programming Kit. http://www.gnu.org/software/glpk/, 2009.

[45] Stoffel, D., and Kunz, W. Equivalence Checking of Arithmetic Circuits on the Arithmetic Bit Level. In *IEEE Trans. on Computer-Aided Design* (May 2004), vol. 23, pp. 586–597.

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

[47] Watanabe, Y., Homma, N., Aoki, T., and Higuchi, T. Application of Symbolic Computer Algebra to Arithmetic Circuit Verification. In *Proc. Intl. Conf. on Computer Design* (2007), pp. 25–32.

[48] Wedler, M., Stoffel, D., Brinkmann, R., and Kunz, W. A Normalization Method for Arithmetic Data-Path Verification. In *IEEE Trans. on Computer-Aided Design* (November 2007), vol. 26, pp. 1909–1922.

[49] Wienand, O., Wedler, M., Stoffel, D., Kunz, W., and Greuel, G. An Algebraic Approach for Proving Data Correctness in Arithmetic Data Paths. In *Proc. Intl. Conf. on Computer-Aided Verification* (July 2008), Springer-Verlag Berlin Heidelberg 2008, pp. 473–486.

[50] Wu, W. T. Mathematics Mechanization. *Front. Comput. Sci. China* (2000), 1–8.

[51] Yang, Z., Ma, G., and Zhang, S. Formal Verification of High-Level Data-Flow Synthesis Designs Using Relational Modeling and Symbolic Computation. *Journal of Integration 43* (January 2010).

[52] Zimmermann, R. *Computer Arithmetic: Principles, Architectures, and VLSI Design.Lecture notes Swiss Federal Institute of Technology (ETH), Zurich, Switzerland.* 1997.