New Solving Techniques for Property Checking of Arithmetic Data Paths

Neue Beweistechniken
für die Eigenschaftsprüfung von arithmetischen Datenpfaden

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

von
M.Sc. Evgeny Pavlenko
geb. in Atbasar, Kasachstan

D 386

Dekan:	Prof. Dr.-Ing. Norbert Wehn
Gutachter:	Prof. Dr.-Ing. Wolfgang Kunz, Technische Universität Kaiserslautern Prof. Dr. Bernd Becker, Albert-Ludwigs-Universität Freiburg
Datum der Disputation:	19 December 2011

Acknowledgments

This thesis is a result of several intensive years spent for PhD research on solving hard arithmetic decision problems at the Electronic Automation Group in the University of Kaiserslautern. I wish to thank all those people who contributed to this research. Without them I could not have completed it and this thesis would not have been possible.

First and foremost, I am greatly indebted to the head of the Electronic Automation Group, Prof. Wolfgang Kunz. He has provided me not only with a very good opportunity to conduct my PhD research in an impressive academic environment but also supported me beginning from my first day I have entered his group. I am especially grateful to Wolfgang for his patience and his great understanding.

I would like to express my gratitude to Prof. Becker for the interest he has shown to review my thesis and for the important and helpful feedbacks.

I owe my deepest gratitude to Markus Wedler who not only served as my direct supervisor but also encouraged and challenged me throughout my PhD research. He has made available his support in a number of ways. His considerable academic guidance, advice and suggestions were very helpful for me. Additionally, I would like to thank Alexander Dreyer and Frank Seelisch for the productive collaboration. Many key ideas presented in this thesis were born during our fruitful discussions with them.

I am also very thankful to Markus, Alexander and Frank as well as to Dominik Stoffel, Max Thalmaier, Sacha Loitz and Oliver Marx for the proof-reading and corrections of my papers or/and this thesis.

I am indebted to my other colleagues at the University of Kaiserslautern, namely Roland Hecker, Minh Duc Nguyen, Binghao Bao, Bernard Schmidt, Joakim Urdahl, Carlos Villarraga, Andreas Christmann, Matthias Legrom, Hans Serr and Carmen Vicente-Fess for friendly atmosphere, interesting talks and support.

Special thanks and appreciation to OneSpin Solutions GmbH, in particular to Raik Brinkmann and Evgeny Karibaev, for the offered formal verification tool and for the provided industrial patterns.

Last but not least, I am very thankful to my parents for their constant support, understanding and caring, especially at that time when I am very far away from them.

Evgeny Pavlenko　　　　Kaiserslautern, December 19th, 2011

To my past and future

Contents

Chapter 1

Introduction

As known from *Moore's law* [Moo65,web], the number of transistors placed on a modern *integrated circuit (IC)* doubles every two years. Therefore, the complexity of *system-on-chip (SoC)* designs increases rapidly. On the other hand, in order to guarantee an error-free operation of SoCs, chips of high quality have to be designed and fabricated.

As a result, the *validation* of digital designs – proving them for correctness in particular – plays a very important role along all steps of a design process. To find bugs and to ensure correctness of a design, three commonly used approaches are applied for validation: *simulation*, *emulation*, and *formal verification*.

In case of simulation a software simulation model of a design is considered. To reflect critical execution traces in this model, input patterns (*stimuli*) are generated either randomly or with regard to some chosen heuristic. Stimuli are applied to the primary inputs of the model. Consequently, some values appear at the primary outputs. These values are observed and checked for equivalence against the expected values. If the produced values are conflicting with the expected ones then something in the design is erroneously implemented. Such a design has to be analyzed to find out the cause of the error. As soon as the error is fixed, the simulation is repeated.

For the case of emulation the overall procedure is almost the same as for simulation. However, the emulation process requires a different design representation. Now, a physical prototype of the design is needed, e.g., implemented by means of a *field-programmable gate array (FPGA)* [BR96].

Simulation-based methods are quite effective. In practice, they are able to detect many design errors. However, these methods have a very important disadvantage when large designs are examined. The number of all possible input stimuli for a model under test grows exponentially with the number of primary inputs in this model. For real industrial designs this number is astronomical. Thus, a simulation of a design with all possible stimuli is an unrealistic scenario for industrial applications. As a consequence such methods may still miss errors in designs. Especially, so-called corner case bugs might be overlooked. A well-known example of a simulation failure is the Intel Pentium bug [Int] encountered in the *floating point unit (FPU)* of the *Intel P5 Pentium* processor. Recently, another bug overlooked by simulation was identified in some chipsets used for the Intel's latest *Sandy*

Bridge processors. The total cost to fix and recover this bug is estimated at about 700 millions of US dollars.

In contrast to simulation and emulation, methods of formal verification can prove the correctness of a design for all possible input values. These methods are implemented on the basis of mathematical proofs. Due to the tremendous improvements of the formal methods they may successfully be applied in practice for real designs. However, in spite of the significant progress made in formal verification within the last few decades, the continuous growing in complexity of modern digital designs makes verification tasks more complex and challenging. As a result, in the design process of digital circuits, about 70% of all costs are nowadays spent for verification. This motivates the research community to develop new effective methods in the field of formal verification.

Generally, the scope of formal verification can be divided into two main tasks, namely *equivalence checking* and *property checking* which are considered in the following sections.

1.1 Equivalence Checking

On its way from a concept to the final fabricated chip, a microelectronic design undergoes different phases. At each phase, the design representation is also different and given at some level of abstraction like *register transfer level (RTL)* or *logic level*.

At every level of abstraction, the design has to be optimized to meet different requirements like timing, power consumption, technology mapping. Moreover, throughout a digital design process, the design representation is also changed from a description of a higher level of abstraction to a description of a lower level. As a result of such optimization and synthesis, *bugs* in a design representation might appear. Typical reasons for these errors are manual designer's interventions, encoding mistakes or bugs caused by EDA software. Therefore, it is extremely important to ensure the correctness of a design after every step of optimization and synthesis. For purposes like this, equivalence checking is used.

The primary goal of equivalence checking is a formal proof that two design representations (potentially of dissimilar internal structure) exhibit identical functional behavior for all input patterns.

A classical verification model for equivalence checking is called *miter* and depicted in Figure 1.1. Here, the functionality of the optimized/synthesized design, also named *implementation*, has to be compared against the known functionality of a *golden model*, also called *specification*. The implementation and the specification have common inputs. However, the outputs O_1, O_2 of the models are computed separately and, in the sequel, are used for comparison by a comparator o.

Basically, there are two main possibilities to solve the task of equivalence checking, namely by means of a structural comparison of two models or by means of reasoning. In the first case, it is necessary to derive some unique representations for the implementation and the specification so that a further straightforward comparison becomes available. To

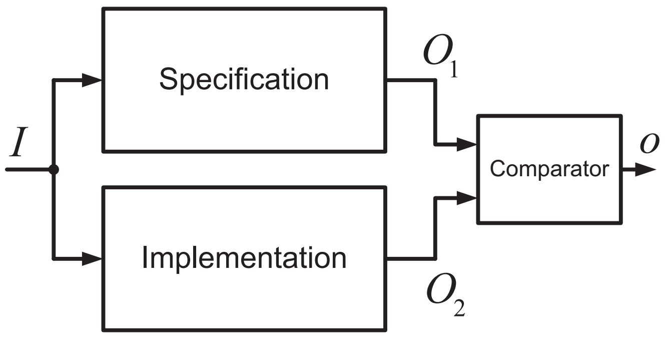


Figure 1.1: Miter scheme for equivalence checking

fulfill this condition in practice, *binary decision diagrams (BDDs)* [Ake78,Bry86] or their modifications [CFM+93, Min93, DBS+94, LS95, BC95, DBR96, SBW98, DB98a, AF07, AF08] are widely used. For instance, an introduction and examples on applications of BDDs in formal verification can be found in [Hu97, SB01, GHB01, KDB+03]. Contrary to a structural approach, the task of equivalence checking can be formulated as a *Boolean satisfiability problem (SAT problem)* which can further be solved with a standard *SAT solver* [MSS99,MMZ+01,GN02,ES03,Pre]. The SAT problem is generated by converting the model of a miter into a *Boolean conjunctive normal form (CNF)*. This process is also know as *bit-blasting*. It is important to note that the task of equivalence checking is not so trivial in case of large industrial designs. Fortunately, there are usually a lot of internal equivalences between the specification and the implementation in a miter model. Exploiting the fact of such internal equivalences may significantly facilitate performing of equivalence checking in practice.

Since a digital design may or may not contain memory elements, it is also common to distinguish between *sequential and combinatorial equivalence checking*.

For combinatorial equivalence checking, many interesting and effective approaches were proposed in the past, e.g., see [Kun93,Bra93,JMF95,Mat96,KK97,SK04,KJW+08].

The proposed techniques usually perform well in practice. However, they may lack robustness as soon as data paths of industrial designs, especially those of them that implement multiplications, have to be considered. In [SK04] the authors developed a method based on reverse engineering to identify as many internal equivalences as possible between a full-custom implementation and its specification. The approach of [KJW+08] provides a special language for a proper modeling of such specifications in a full-custom design flow.

In sequential equivalence checking additionally to all possible inputs evaluations, all reachable states of the design have to be taken into account. In other words, a *reachability analysis* is necessary. This might result in a *state explosion problem*, since the number of reachable states increases exponentially with the number of memory elements used in a design. Thus, the techniques of sequential equivalence checking, for example such as [HCCG96, SK97, vE98, SWWK04], are more complicated and, unfortunately, not yet mature in industrial practice.

1.2 Property Checking

At earlier stages of a design process, a digital hardware design is described in terms of an informal specification. On the base of this specification, an RTL model of the design is created. At this point, it is very important to assure that such a model performs correctly in accordance to the specification requirements. Otherwise, redesigns and, thereby, increase in costs of the whole design flow become unavoidable.

The primary goal for application of property checking in practice is to prove that a created RTL model captures all the required properties as defined by the specification. Also, by doing property checking, a designer gets a better understanding of the design. Moreover, property checking is commonly used for debugging of a design. For a more thorough review on application of formal property checking in industry, the reader is referred to [BJW04].

In fact, to implement property checking, three major components are needed. They are as follows: a mathematical model of the RTL design in question, an appropriate language to formulate properties, and a method to efficiently solve the task of property checking.

In the past, fully automatic approaches of *model checking* [CGP99] were under intensive investigation. In [CE81] *computational tree logic (CTL)* was proposed to formulate properties for a *Kripke model* which is very similar to a *state transition graph* generated from a *finite state machine (FSM)* of the design. For this model every property p is formulated as an CTL formula f_p. The set of all states S_f has to be computed where the formula f_p is valid. The property p is then verified if and only if it holds that $S_0 \subseteq S_f$, where S_0 forms the set of initial states in the Kripke model under consideration. In this approach an explicit representation of computed states is used. As a result, this approach suffers from the state explosion problem in case of large designs. This problem is alleviated in *symbolic model checking (SMC)* [McM93], where BDDs are applied to create an implicit representation for the set of states. As shown in [BCL$^+$94], properties can successfully be verified for designs with up to 10^{120} states. However, the number of states in many industrial designs is even larger so that SMC is not capable to deal with them.

1.2.1 SAT-based Property Checking

Nowadays, so-called *SAT-based property checking* has become a workhorse in the modern property checking flow. The basic idea behind this method is to represent the task of property checking as a SAT problem, i.e., generate a plain CNF by bit-blasting the model and the property under interest so that a standard SAT solver can be applied. A SAT solver tries to prove that the CNF is satisfiable, i.e., the solver tries to find a value assignment which violates the property. If such an evaluation is discovered then, for a further analysis why the property fails for this design, a *counterexample* may be generated from the satisfiable assignment. On the contrary, if a SAT solver is able to prove that the CNF is unsatisfiable for all possible evaluations of the CNF then the property holds for this design. At present, SAT-based techniques are key in modern RTL property checking flows, see Figure 1.2.

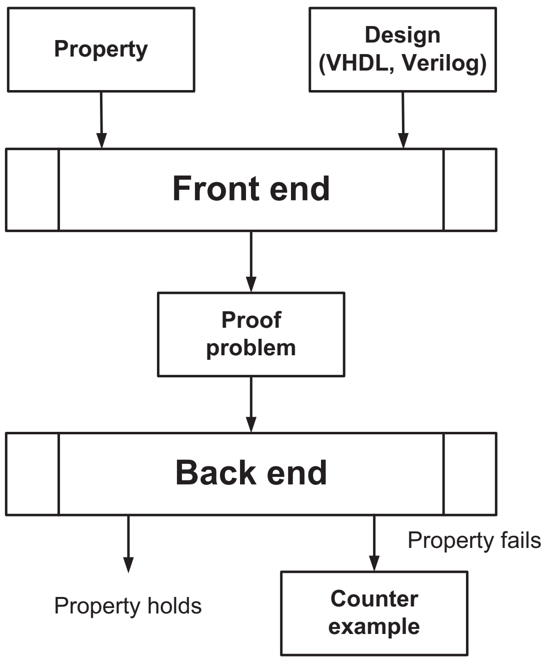


Figure 1.2: Typical flow for RTL property checking

A significant improvement for property checking was proposed in [BCCZ99]. According to this approach, the combinatorial part of an FSM, also known as a *time frame*, is unrolled into an combinatorial instance called *iterative circuit array*. The unrolled time frames are successively connected from the initial state up to some k next states so that the primary outputs of a time frame i feed the primary inputs of the time frame $(i + 1)$. Further, the iterative circuit array is appended with a combinatorial logic describing a property. This model is then translated into a SAT instance and handed over to a SAT solver. This approach is called *bounded model checking (BMC)*. It is a lot more efficient than the method of SMC. However, BMC has a substantial drawback. BMC may sometimes lead to *false positives*. This happens if a property p will fail in a state s which is beyond the maximum time frame k for a bounded model checker.

To overcome this problem, *interval property checking (IPC)* was introduced. In IPC, time frames are unrolled starting from an arbitrary state. This approach demonstrated its high effectiveness in practice, e.g., in an RTL property checker like [One], but here another problem called *false negative* might appear. Since the initial state for the unrolled circuit is unconstrained, this state may not always be reachable from the initial state of the design. Thus, for the scenario when a property fails in such an unreachable state, the property checker erroneously proves that the property fails for the design. Therefore, one has to precisely specify the restrictions on the environment and states of the design observed for an actual property. The work of [NSWK05] is an approach to compute such reachability constraints automatically.

In modern SoC design flows, it is a common strategy to integrate different modules, called *intellectual property (IP)* blocks, on a single chip. Besides functional verification for separate IP blocks, it is also necessary to verify communications between these blocks. The communication issues between IP modules in a SoC are specified with protocols. Therefore, communicational verification for a SoC is named *protocol compliance verification* [NTW$^+$08, NTM$^+$09].

In conclusion to this section it should be noted that in spite of considerable improvements achieved for property checking in the last decades, a formal proof for correctness of an RTL data-path design with arithmetic circuits like, e.g., multiplier units may still be a challenging task. In [WSK05, WSBK07] an efficient method was presented to verify a data path at the *arithmetic bit level (ABL)*. An ABL description of a design can easily be provided with a front end of a property checker. In industrial practice, however, some portions of such an ABL description may be missing, since parts of a data path may be optimized at the logic level. The aspects of solving hard arithmetic problems with different approaches in formal verification is briefly discussed in the next section.

1.3 Motivation and Thesis Overview

Modern design flows for SoCs pursue a correctness-by-integration strategy when verifying the functionality of the overall system. This requires high quality designs for the individual modules that are supposed to be integrated into the SoC. Traditional simulation-based verification techniques are reaching their capacity limits rapidly. This motivates the application of highly automated property checking techniques.

A promising approach is called interval property checking (IPC). It is mostly based on *satisfiability solving (SAT)* and *SAT modulo theory solving (SMT)*. IPC has the capacity to handle almost all types of modules that can be found in today's SoCs. Nonetheless, a few pathological cases remain that sometimes limit its application in industrial practice. In particular, data paths are often a challenge. This is true, especially, if not only the correctness of the control flow but also the correctness of the computed data needs to be proved.

For complex arithmetic data paths, simulation is, therefore, still prevailing in industrial verification environments. This is due to the inability of standard proving procedures to handle arithmetic functions. Especially, multiplication – as it is part of nearly all data paths for signal processing applications – has remained a severe problem for standard tools. This problem has stimulated a lot of research on specialized proof methods with focus on arithmetic. The main achievements and challenges of formal methods for verification of an arithmetic data path are briefly reviewed in the next section.

1.3.1 Related Work and Challenges

There exists a large variety of techniques tackling arithmetic circuit verification in different ways. Word-level decision diagrams like *BMDs [BC95] have been investigated that promise a compact canonical representation for arithmetic functions. By lack of robust synthesis routines to derive these diagrams from bit-level implementations, *BMDs are, however, hardly used in RTL property checking. For example, *Hamaguchi's method* for *BMD synthesis [HMY95] suffers from diagram blow-up in case of faulty circuits as noted by Wefel and Molitor [WM00]. Generating diagrams from bit-level specifications has remained an unresolved issue also for more recent developments such as *Taylor expansion diagrams (TED)* [CZKR02], *LTED* [AF07], *Modular-HED* [AF08].

Another intensive research area of the last several decades was SAT solving [MSS99, MMZ+01, GN02, ES03, Pre]. The techniques of SAT solving play a major role in almost all modern verification tools. For control-intensive modules of SoC designs SAT solvers have shown to be adequate proof engines when verifying their correctness. However, data paths including complex arithmetic blocks such as multiplication still remain a bottleneck for SAT-based property checkers.

As SAT-based techniques have become the predominant proof methods in formal verification, significant efforts were made to integrate SAT with solvers for other domains. The integration of SAT and *integer linear programming (ILP)* techniques leads to hybrid solvers like [CK03, ABC+02]. However, ILP turns out to be unsuitable for RTL property checking because non-linear arithmetic functions need to be handled. Unfortunately, even simple multiplication falls into this category.

More recently, SMT solvers have gained significant attention [GHN+04]. These solvers integrate different theories into a unified *DPLL-style* decision procedure. For proving functional correctness of arithmetic data paths the theory of fixed-sized *bit vectors (BV)* becomes a natural choice. The *quantifier-free logic over fixed-sized bit vectors (QF-BV)* is a logic that facilitates interpretation of bit vector functions with respect to their semantics. Different research groups developed SMT solvers for the QF-BV category, e.g., see [DdM06, dMB07, BH08, BB09, BCF+07, GD07, STP]. These tools demonstrated significantly better performance in comparison to pure SAT. However, they still lack capacity to prove unsatisfiable formulas as they are derived from arithmetic data-path verification in industry.

For equivalence checking of arithmetic RTL circuits, especially multipliers, a technique based on rewriting was proposed in [VVSA07]. A database of rewrite rules is provided to support a large number of widely used multiplier implementation schemes. However, for non-standard implementations the approach requires updating the database manually and is, thus, not fully automatic. Fully automatic techniques for equivalence checking and debugging of arithmetic data paths are provided in [SK04, STAF09]. These techniques extract arithmetic bit level information from low level gate net lists. They consider the data path to be clearly separated from control logic which in high performance RTL designs is often not the case. This renders integration of the techniques into a general purpose SMT solver with a property checking application scenario exceedingly difficult.

An alternative direction of research focuses on techniques operating at the *arithmetic bit level (ABL)* description of a data path. In [SK04] an extraction technique is presented that automatically extracts ABL information from optimized gate netlists of arithmetic circuits after synthesis. This approach is mainly designed for application in equivalence checking, and its arithmetic reasoning on the ABL is restricted to a single addition network. For RTL property checking, however, global reasoning over several arithmetic components is required. In [WSBK07] a generalization of the ABL description is introduced and a normalization calculus for property checking was presented. All ABL information needed for property checking can usually be obtained directly from the RTL description. Unfortunately, this may change in full-custom design flows where ABL information may no longer be available at the RT-level. In order to apply ABL techniques in a completely full-custom design flow, a description language for arithmetic circuits is introduced in [KJW$^+$08] that captures the necessary ABL information. The methods used for the arithmetic proofs are similar to the normalization approach of [WSBK07].

Lately, methods of symbolic computer algebra have gained attention for verification of arithmetic circuits. In these methods, *multivariate polynomials* are derived from bit-vector functions of a data path. For example, Shekhar et al. [SKEG05, SKE06, SKE07] demonstrated an equivalence test for polynomial representations of arithmetic functions. Here, a word-level description of a data-path design is required. Unfortunately, such a description is not always available in practice, since some parts of an RTL design can be optimized at the gate level. These parts are called *custom-designed components*. A typical example of this component in a multiplier unit is a Booth encoder.

In [WHAH07] a technique was proposed that exploits an ABL description to generate multivariate polynomials and computes a Gröbner basis thereof. For a proof goal – represented as an additional polynomial – a normal form with respect to this Gröbner basis is calculated. In summary, this technique shows a good performance for solving arithmetic verification problems. However, the obligatory computation of a Gröbner basis might be too expensive for some RTL designs in practice and, thus, not always applicable.

As opposed to [WHAH07], Wienand et al. [WWS$^+$08] proved that a polynomial system derived from an ABL description of a design and the property in question is by construction a Gröbner basis for some chosen monomial ordering. Similar to [SKEG05, SKE06, SKE07], this approach also requires arithmetic parts of data-path design be available at the word level or at the ABL. Unfortunately, it is not the case for a custom design methodology, when some portions of a design are optimized below ABL.

1.3.2 Objective and Outline of this Thesis

The main objective of this thesis is to increase the capacity of the ABL-based techniques and the approaches of computer algebra that enables more robust and efficient solving of hard arithmetic problems originating from property checking of a data-path design. Additionally to that a robust SMT solver for the QF-BV category has to be developed.

This thesis is organized as follows.

Chapter 2 provides the necessary terminology and fundamentals that are related to the topic of this thesis.

Chapter 3 introduces the concept of the arithmetic bit level. The main ABL components such as a partial product generator, an addition network, and a comparator are also defined in this chapter. Moreover, the ABL-based techniques and their applications in domains of formal verification and debugging are reviewed.

Chapter 4 describes an improved algebraic approach based on the theory of Gröbner basis. This chapter starts with a brief summary of the previous works related to this topic and continues with a motivation for the improvements of the algebraic approach. Its mathematical background is described in the sequel. At first, modeling of arithmetic decision problems is discussed and illustrated by means of a few examples. Thereafter, a mathematical proof is provided for the algebraic approach to solve these decision problems. This chapter ends with a didactic example demonstrating application of the approach to solve arithmetic verification problems in RTL property checking.

Chapter 5 proposes two approaches to convert a gate netlist of a custom-designed component into a functionally equivalent ABL description or a system of multivariate polynomials over Boolean variables. At the beginning, this chapter familiarizes the reader with the problem for ABL-based techniques to handle custom-designed components of an RTL design. This motivation is then followed by the explanations of methods how an ABL description can be generated from such components. One of these methods improves robustness of ABL normalization. The next method is considered for the algebraic approach. At the end of this chapter, the experimental results are presented. Here, the ABL normalization technique strengthened with the extraction method is compared against contemporary SMT solvers for the QF-BV category.

Chapter 6 develops a new QF-BV SMT solver called *STABLE*. The solver integrates two recently developed techniques, namely the extraction method and the enhanced approach of computer algebra. In addition to that, some rewriting and simplification mechanisms are designed for the front-end engine of the SMT solver. A standard SAT solver is used as a back end of STABLE. The performance of STABLE was tested against other SMT solvers – winners of SMT competitions in the last few years. The experimental data of the tests close this chapter.

Chapter 7 concludes this thesis and provides discussion on the future research.

Chapter 2

Fundamentals

This chapter provides some fundamental mathematical concepts which are of relevance for this thesis. All important symbolic notations used in later chapters will also be defined here. Since this chapter is rather a brief review of underlying notions, then, in case more detailed explanations are needed, the reader may refer to standard textbooks, e.g., on switching theory [AS06], discrete mathematics [GT96], Boolean logic [Bro90], and algorithms of logic synthesis and verification [Mic94, HS96].

2.1 Relations

This section contains some important definitions for a set of elements and explains some particular relations over the elements inside a set.

Let lowercase letters denote elements of a set whereas uppercase letters denote a set of elements. The notation $a \in A$ means that the element a belongs to the set A. Conversely, $a \notin A$ stands for the case when a is not a part of A. For example, the set of integers is denoted as $\mathbb{Z}$ and expression $i \in \mathbb{Z}$ is valid for any integer i. If the set B consists of some elements $a_0, \ldots, a_{n-1}$ then one writes $B = \{a_0, \ldots, a_{n-1}\}$. For finite sets, the number of elements in the set B builds its *cardinality* and is denoted as $|B| = n$. Some important *relations* can be defined on sets.

Definition 2.1. *(***Binary Relation***) For sets A and B, a set of pairs of elements (a, b) defines a binary relation R from A to B if each $a \in A$ and each $b \in B$. The expression $(a, b) \in R$ can syntactically also be written as aRb or $R \supseteq \{(a, b)\}$.*

In this thesis only binary relations are considered. For reason of simplicity the adjective *binary* is omitted and just the word *relation* is used in the sequel. Two extreme cases are available for relations, namely the so-called *universal relation* and the *empty relation*. The universal relation is the *Cartesian product* $A \times B$ that consists of all possible pairs (a, b) with $a \in A$ and $b \in B$. The empty relation, denoted as $\emptyset$ or $\{\}$, contains no pairs of elements.

Definition 2.2. *(***Inverse Relation***) For a relation R from A to B the inverse relation R^{-1} from B to A is defined by $R^{-1} = \{(b, a) : (a, b) \in R\}$.*

A relation R over elements of a single set S may have some interesting properties as follows:

- *reflexivity*: $(a, a) \in R$ for all $a \in S$,
- *symmetry*: $(a, b) \in R$ implies $(b, a) \in R$ for all $a, b \in S$,
- *anti-symmetry*: $(a, b) \in R$ and $(b, a) \in R$ imply $a = b$ for all $a, b \in S$,
- *transitivity*: $(a, b) \in R$ and $(b, c) \in R$ imply $(a, c) \in R$ for all $a, b, c \in S$.

A relation R on a set S is named an *equivalence relation* if and only if (*iff*) it is reflexive, symmetric and transitive. An equivalence relation R splits elements of the set S into subsets $S_0, S_1, \ldots, S_{n-1}$, called the *equivalence classes*, such that $S = \bigcup_{i=0}^{n-1} S_i$ and for any $i \neq j$ it is valid that $S_i \bigcap S_j = \emptyset$. In other words, the equivalence relation R builds the *partition* $\pi = \{S_0, S_1, \ldots, S_{n-1}\}$ on the set S. In this case, S_i is also known as a *block* of the partition. If two elements a and b belong to the same block then one says a is equivalent to b and writes $a \equiv b$.

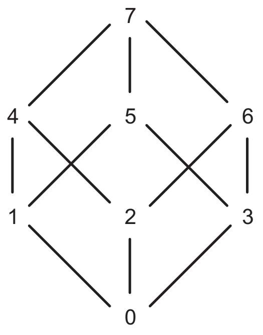


Figure 2.1: Hasse diagram

When the relation R on the set S is reflexive, anti-symmetric and transitive then this is a *partial-order* relation which can be also thought as a *"less than or equal to"* relation. A set S with such a relation R is called a *partially-ordered* set and is usually represented in terms of a *Hasse diagram.*

Example 2.1. *(***Hasse Diagram***) For the partially-ordered set $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$, the Hasse diagram is depicted in Figure 2.1.*

In a partially-ordered set S with the relation R, the following essential elements may be identified:

- the *least element* is the element $a \in S$, where for any $z \in S$ holds $(a, z) \in R$,
- the *greatest element* is the element $b \in S$, where for any $z \in S$ holds $(z, b) \in R$,
- a *lower bound* of the elements x and y is an element c such that $(c, x) \in R$ and $(c, y) \in R$, where $x, y, c \in S$,
- an *upper bound* of the elements x and y is an element d such that $(x, d) \in R$ and $(y, d) \in R$, where $x, y, d \in S$,
- the *greatest lower bound* of two elements x and y is such a *lower bound* c' that $(c_i, c') \in R$ for any *lower bound* c_i of x and y, where $x, y, c', c_i \in S$,
- the *lowest upper bound* of two elements x and y is such an *upper bound* d' that $(d', d_i) \in R$ for any *upper bound* d_i of x and y, where $x, y, d', d_i \in S$.

The operations to calculate the greatest lower bound and the lowest upper bound are designated with the symbols $\cdot$ and $+$, respectively. Those partially-ordered sets where for any pair of elements there exist the greatest lower bound and the lowest upper bound are called a *lattice*. Usually, the greatest element is denoted as 1 and the least element is denoted as 0. For instance, the lattice is represented with the Hasse diagram in Figure 2.1. A lattice is a mathematical structure on the base of which *Boolean algebra* can be defined.

2.2 Boolean Algebra

Definition 2.3. *(***Boolean Algebra***) A Boolean lattice or Boolean algebra on the set S is a complementary and distributive lattice where the following two laws are valid:*

- *complement: for any $a \in S$ there exists an $\overline{a} \in S$ such that $a \cdot \overline{a} = 0$ and $a + \overline{a} = 1$,*
- *distributivity: for any a, b, $c \in S$ it holds that $a \cdot (b + c) = a \cdot b + a \cdot c$.*

As it was shown by Shannon in [Sha38], the behavior of switching circuits is well described by means of two-valued Boolean algebra $\mathbb{B} = \{0, 1\}$. Nowadays, Boolean algebra plays a very important role in switching theory, digital circuits design and formal verification where any $a_i \in \mathbb{B}$ is often called as a *bit* and a concatenated sequence of the bits $(a_0, \dots, a_n)$ is known as a *bit vector*.

In addition, a Boolean algebra fulfills the following rules:

- *commutativity*: $a + b = b + a; \quad a \cdot b = b \cdot a,$
- *associativity*: $a + (b + c) = (a + b) + c; \quad a \cdot (b \cdot c) = (a \cdot b) \cdot c,$
- *idempotence*: $a + a = a; \quad a \cdot a = a,$
- *absorption*: $a + (a \cdot b) = a \cdot (a + b),$
- *identities*: $a + 0 = a; \quad a + 1 = 1; \quad a \cdot 0 = 0; \quad a \cdot 1 = a,$
- *De Morgan's law*: $\overline{(a + b)} = \overline{a} \cdot \overline{b}; \quad \overline{(a \cdot b)} = \overline{a} + \overline{b},$
- *resolution*: $(a + b) \cdot (c + \overline{b}) = (a + b) \cdot (c + \overline{b}) \cdot (a + c),$
- *consensus*: $a \cdot b + c \cdot \overline{b} = a \cdot b + c \cdot \overline{b} + a \cdot c.$

Further logic operations which concern this thesis can be defined as follows:

- *exclusive or (XOR)*: $a \oplus b = (a + b) \cdot (\overline{a} + \overline{b}),$
- *implication*: $a \rightarrow b = (\overline{a} + b),$
- *equivalence*: $a \leftrightarrow b = \overline{(a \oplus b)},$
- *if then else (ITE)*: $ite(a, b, c) = (a \cdot b + \overline{a} \cdot c).$

In the literature, the operations $\cdot$ and $+$ are also referred to as a *product* and a *sum*, respectively. Additionally, the complement operation as well as the operations $\cdot$ and $+$ are frequently designated with logic symbols, namely $\neg$ (*negation* or *NOT*), $\wedge$ (*logic AND* or *conjunction*) and $\vee$ (*logic OR* or *disjunction*).

2.3 Graphs

Relations on a set of elements are well visualized by an abstract representation called *graph*. Moreover, graphs are a proper way to formulate, analyze, and solve many technical problems.

Definition 2.4. *(***Graph***) A graph G is a pair (V, E), where $V = \{v_0, \ldots, v_n\}$ is a nonempty set of vertices (nodes) and $E = \{e_0, \ldots, e_k\}$ is a set of edges. For a given set of the nodes V, the set of the edges E is defined as a set of pairs of the nodes, where the edge between the node v_i and the node v_j is denoted with $e_t = (v_i, v_j)$. Here, v_i and v_j are called the adjacent nodes. Moreover, v_i is named the immediate predecessor of v_j and v_j is the immediate successor of v_i. Two edges are adjacent if they have a common node.*

Often for the edge (v_i, v_j), the pair of the elements v_i and v_j can be considered to be ordered from v_i to v_j. If all edges of the graph are ordered then this graph is *directed*, otherwise, it is called an *undirected* graph.

For a graph G we call $G' = (V', E')$ a *subgraph* of G if and only if it holds that $V' \subseteq V$ and $E' \subseteq E$. The *degree* of a node is the number of edges connected to this node. In case of an undirected graph, this is the number of all connected edges. In a directed graph it is distinguished between the incoming edges, also know as a *fanin*, and the outgoing edges called *fanout*. While traversing a graph the following notions become necessary:

- *loop* is an edge with the same start and end node, i.e., (v_i, v_i),
- *walk* is a finite sequence of adjacent edges, e.g., $(v_0, v_1), (v_1, v_2), (v_2, v_3)$,
- *trail* is a such walk throughout which each edge appears at most once,
- *path* is a such walk throughout which each node appears at most once,
- *cycle* or *closed path* is such a sequence $(v_0, v_1), \ldots, (v_{n-1}, v_n), (v_n, v_0)$, where $(v_0, v_1), \ldots, (v_{n-1}, v_n)$ is a path.

Hence, an *acyclic* graph is a graph without any cycles. An important class of the graphs is a *directed acyclic graph* or a *digraph (DAG)*. DAGs are widely used in computer science, e.g., to represent Boolean functions, combinatorial circuits and many others.

2.4 Representation of Boolean Functions

Boolean functions represent a *mapping* in the *Boolean space* $\mathbb{B}^k = \{0,1\}^k$. For n inputs and m outputs, a *Boolean function* is formally defined as a mapping $F : \mathbb{B}^n \mapsto \mathbb{B}^m$, where $n, m \in \mathbb{N}$.

Definition 2.5. *(***Function***) A binary relation $F \subseteq A \times B$ is called a function if and only if the following axioms are valid:*

- *right-unique (also called right-definite or functional): for all $a \in A$ and $b, c \in B$ so that $(a,b) \in F$ and $(a,c) \in F \Rightarrow b = c$,*
- *left-total: for all $a \in A$ there exists $b \in B$ so that $(a,b) \in F$.*

If the values for all inputs are known, the function is called *completely specified*. However, in practice for some inputs, the values might be unknown; this is called *don't care* which is designated with the symbol $*$. In this case, the Boolean function becomes *incompletely specified* $F : \{0,1\}^n \mapsto \{0,1,*\}^m$.

A common way to express a Boolean function is a *Boolean formula* or a *Boolean expression* which depends on a set of Boolean variables $X = \{x_0, \ldots, x_n\}$. Let a *literal* be a variable or its complement. Formulas can recursively be defined in terms of the *factored form* [Mic94].

Definition 2.6. *(***Factored Form***) The factored form is one of the following Boolean expressions:*

- *Boolean constant 0 or 1,*
- *literal,*
- *conjunction of factored forms,*
- *disjunction of factored forms.*

Boolean expressions are a customary readable mathematical form for humans, however, they are not always suited well for the applications in specific domains like logic synthesis, optimization, verification. In fact, there are different ways to represent Boolean functions, e.g., *truth table*, *Karnaugh-Veitch maps (KV-map)*, *conjunctive normal form (CNF)*, *disjunctive normal form (DNF)*, *binary decision diagram (BDD)*, *Boolean network*, *switching circuit* and others. Some of these representations are briefly described in the next sections.

2.4.1 Truth Table

For a function f a *truth table* is a list of all enumerations of all possible input values mapped to the output values of this function.

a_0	a_1	a_2	$f(a_0, a_1, a_2)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 2.1: Truth table

Example 2.2. *(***Truth Table***) The truth table of the function $f = (a_0 \wedge a_1) \vee (a_1 \wedge a_2) \vee (\overline{a_1} \wedge \overline{a_2})$ is depicted in Table 2.1.*

Here, each line contains a unique assignment for the input variables as well as the corresponding value for the function. The set of the input assignments when f becomes 1 is known as *ON-set* of f. Conversely, the *OFF-set* consists of all the input assignments when f is 0. With a predefined order of the input assignments in a truth table, this table becomes a *canonical* representation. The canonicity of a Boolean data structure is a very important property as it solves a fundamental problem in formal verification, i.e., a decision problem whether two Boolean functions are functionally equivalent. However, since the size of a truth table grows exponentially with the number n of the input variables, namely 2^n, in practice such a representation of Boolean functions can only be applied for functions of small size.

2.4.2 Binary Decision Diagram

Binary decision diagrams (BDD) are a popular way to manipulate Boolean functions in *computer-aided design (CAD)* and verification. BDDs were first introduced in [Lee59], later studied in [Ake78] and intensively investigated in [Bry86].

The underlying principle for the composition of BDDs from Boolean functions is a recursive decomposition of these functions based on *Shannon's expansion* theorem.

Definition 2.7. *(***Shannon's Expansion***) A Boolean function $f(x_0, \dots, x_n)$ can be represented in the next form:*

$$f = x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}$$

with any $i \in [0 : n]$, where

- *$f_{x_i} = f(x_0, \dots, x_i = 1, \dots, x_n)$ is a cofactor of f with respect to x_i,*
- *$f_{\overline{x_i}} = f(x_0, \dots, x_i = 0, \dots, x_n)$ is a cofactor of f with respect to $\overline{x_i}$.*

A BDD with some predefined order on the variables is called an *ordered binary decision diagram (OBDD)*.

Definition 2.8. *(***OBDD***) An OBDD is a digraph (V, E) with one root node, i.e., the node without any predecessors. Each terminal node $v_j \in V$ has as an attribute $value(v_j) \in \mathbb{B}$. Each non-terminal node $v_i \in V$ has three attributes:*

- *an index(v_i) $\in \{0, \dots, n\}$ mapped to one of the input variables $\{x_0, \dots, x_n\}$,*
- *two immediate successors low(v_i), high(v_i) $\in V$ such that index(v_i) $<$ index($low(v_i)$) and index(v_i) $<$ index($high(v_i)$).*

Definition 2.9. *(***Semantic of an OBDD***) An OBDD (V, E) with the root node v represents a Boolean function f^v as follows:*

- *if v is a terminal node with $value(v) = 0$ then $f^v = 0$,*
- *if v is a terminal node with $value(v) = 1$ then $f^v = 1$,*
- *if v is a non-terminal node with $index(v) = i$ then $f^v = x_i \cdot f^{high(v)} + \overline{x_i} \cdot f^{low(v)}$.*

It is important to note that an OBDD is not a canonical representation of the function because for some nodes in the OBDD their subgraphs can be (*isomorphic*) or even identical, i.e., they define the same subfunction. However, such OBDDs can be reduced through the elimination of the isomorphism in the subgraphs. This results in a *reduced OBDD (ROBDD)* which is now of a unique representation for the function if the specific ordering of the variables is fixed.

Example 2.3. *(***OBDD and ROBDD***) For the function f from Example 2.2, the OBDD and the ROBDD with the variable ordering a_0, a_1, a_2 are depicted in Figure 2.2.*

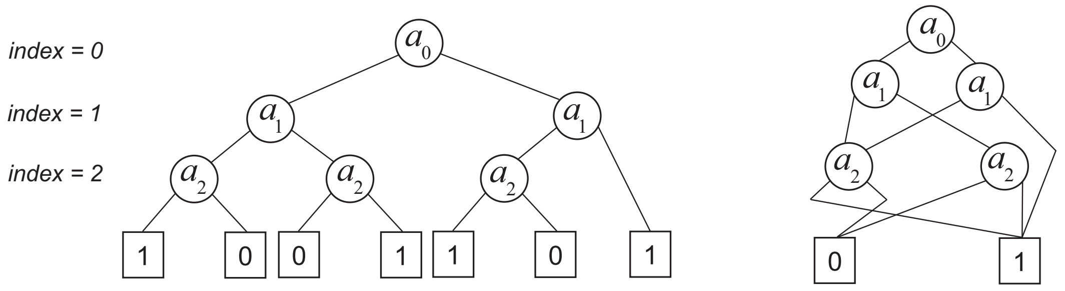


Figure 2.2: Example of OBDD and ROBDD

Besides being a compact representation of a Boolean function, binary decision diagrams can also be manipulated with effective algorithms as it was shown by Bryant in [Bry86].

The size of an OBDD, expressed in terms of the number of nodes in the OBDD, depends not only on the function under the consideration but also on the chosen variable ordering. For some functions, however, the complexity of BDDs is always exponential as, e.g., in case of a bit-vector multiplication.

The Shannon's expansion is not the only decomposition useful for building BDDs. For example, *ordered functional decision diagrams (OFDD)* [KSR92, BD94] are based on the *Reed-Muller decomposition*, also known as a *positive Davio decomposition*.

Definition 2.10. *(***Reed-Muller Decomposition***)*

$$f(x_0, \ldots, x_n) = f_{\overline{x_i}} \oplus x_i \cdot \frac{\partial f}{\partial x_i}$$

where $\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\overline{x_i}}$ *is known as a Boolean derivative of* f*.*

The recursive application of the Reed-Muller decomposition with respect to all variables $\{x_0, \ldots, x_n\}$ on some Boolean formula f converts this formula into a so-called *Boolean (also called Zhegalkin) polynomial.* In overall, the complexity of OFDDs is similar to that of BDDs.

In the past, depending on the application domain and the problem of interest to be solved, many other variants of BDDs were suggested, for instance, *MTBDD* [CFM$^+$93], *ZDD* [Min93,Mis01], *OKFDD* [DBS$^+$94,BDT95,DBJ98,DB98b], *EVBDD* [LS95], *BMD* [BC95], *K*BMD* [DBR96], *LTED* [AF07], *Modular-HED* [AF08]. As the depth of all these variants is beyond the scope of this thesis, the interested reader is referred to the above-mentioned literature.

2.4.3 CNF and DNF

In principle, the *conjunctive normal form (CNF)* and the *disjunctive normal form (DNF)* are Boolean expressions written in special forms. Let us define these forms and their components for some Boolean function $f : \mathbb{B}^n \mapsto \mathbb{B}$.

Definition 2.11. *(***Monomial***) A monomial, also referred to as a product term, is one of the following:*

- *Boolean constant 1,*
- *a literal,*
- *a conjunction of literals such that every variable appears at most once.*

Definition 2.12. *(***Minterm***) A minterm is a monomial that contains all variables of the function* f*.*

Definition 2.13. *(***DNF***) A DNF, also referred to as a sum of products (SOP), is one of the subsequent formulas:*

- *Boolean constant 0,*
- *a monomial,*
- *a disjunction of monomials.*

If a *DNF* consists only of the minterms of the function f then such *DNF* is a canonical representation of f and, therefore, designated as *CDNF*.

Definition 2.14. *(**Clause**) A clause, also referred to as a sum term, is one of the following items:*

- *Boolean constant 0,*
- *a literal,*
- *a disjunction of literals such that every variable appears at most once.*

Definition 2.15. *(**Maxterm**) A maxterm is a clause that contains all variables of the function f.*

Definition 2.16. *(**CNF**) A CNF, also referred to as a product of sums (POS), is one of the subsequent formulas:*

- *Boolean constant 1,*
- *a clause,*
- *a conjunction of clauses.*

Analogously to a *CDNF*, there exists a canonical *CNF* (*CCNF*) which comprises maxterms only. Note, both *CDNF* and *CCNF* can easily be derived from a truth table. The *OFF-set* of the table represents the *CCNF*, conversely, the *ON-set* represents the *CDNF*.

Example 2.4. *(**CCNF and CDNF**) The canonical normal forms for the function f of Table 2.1 are as follows:*

- $f_{CCNF} = (a_0 \vee a_1 \vee \overline{a_2}) \wedge (a_0 \vee \overline{a_1} \vee a_2) \wedge (\overline{a_0} \vee a_1 \vee \overline{a_2})$,
- $f_{CDNF} = (\overline{a_0} \wedge \overline{a_1} \wedge \overline{a_2}) \vee (\overline{a_0} \wedge a_1 \wedge a_2) \vee (a_0 \wedge \overline{a_1} \wedge \overline{a_2}) \vee (a_0 \wedge a_1 \wedge \overline{a_2}) \vee (a_0 \wedge a_1 \wedge a_2)$.

Again, the canonicity is a very important property of a *CDNF* and a *CCNF*. Unfortunately, the complexity of both is exponential in the number of variables. Nevertheless, *CNFs* are used in practice as a fundamental data structure to represent Boolean functions for *SAT solving*, see, e.g., Section 2.5.

2.4.4 Boolean Networks

The structural and behavioral representation of a Boolean function can be expressed in terms of a *Boolean network*.

Definition 2.17. *(**Boolean Network**) A Boolean network is a DAG $N = (V, E, S, F)$, where:*

- *$V = \{I \cup O \cup V'\}$ is a set of the nodes in the graph such that*

  - ∗ *I is a set of primary inputs, i.e., nodes without predecessors,*
  - ∗ *O is a set of primary outputs, i.e., nodes without successors,*

- *$F = \{f_0, \ldots, f_n\}$ is a set of Boolean functions such that every f_i is mapped to some node $v_i \in \{O \cup V'\}$,*
- *$S = \{s_0, \ldots, s_k\}$ is a set of variables such that every s_j is mapped to some $v_j \in V$,*
- *E is a set of edges such that $(v_x, v_y) \in E$ means that the function f_y depends on the variable s_x.*

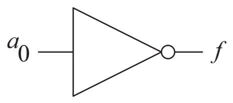
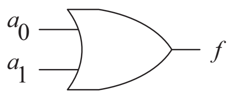
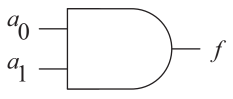
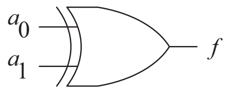
Gate symbol	Name	Boolean function	Truth table
	NOT	$f = \overline{a_0}$	$\begin{array}{c\|c} a_0 & f \\ \hline 1 & 0 \\ 0 & 1 \end{array}$
	OR	$f = a_0 \vee a_1$	$\begin{array}{cc\|c} a_0 & a_1 & f \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array}$
	AND	$f = a_0 \wedge a_1$	$\begin{array}{cc\|c} a_0 & a_1 & f \\ \hline 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{array}$
	XOR	$f = a_0 \oplus a_1$	$\begin{array}{cc\|c} a_0 & a_1 & f \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array}$

Table 2.2: Logic gates. Names and semantics

In principle, a Boolean formula may contain any arbitrary Boolean operators, but, in practice, these operators are fixed by some cell library. Usually, the cells types available

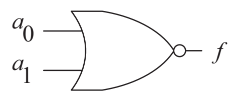
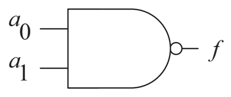
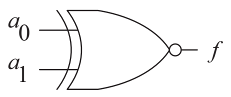
Gate symbol	Name	Boolean function	Truth table
	NOR	$f = \overline{a_0 \vee a_1}$	$\begin{array}{cc\|c} a_0 & a_1 & f \\ \hline 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{array}$
	NAND	$f = \overline{a_0 \wedge a_1}$	$\begin{array}{cc\|c} a_0 & a_1 & f \\ \hline 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array}$
	XNOR	$f = \overline{a_0 \oplus a_1}$	$\begin{array}{cc\|c} a_0 & a_1 & f \\ \hline 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{array}$

Table 2.3: Logic gates. Names and semantics, continued

in a library are dictated by the technology used to fabricate the digital circuit. Sometimes, the cells are a collection of the primitive Boolean functions. In this case, the cell is called a *gate* and the Boolean network is named as a *gate netlist* or a *bit-level description*. There also exist technology independent gate netlists in practice. The symbols for gates are defined by standards *ANSI/IEEE Std 91-1984* and *ANSI/IEEE Std 91a-1991* [AIS86]. Some common representations of the symbols, also used further throughout this thesis, are collected in Table 2.2 and in Table 2.3.

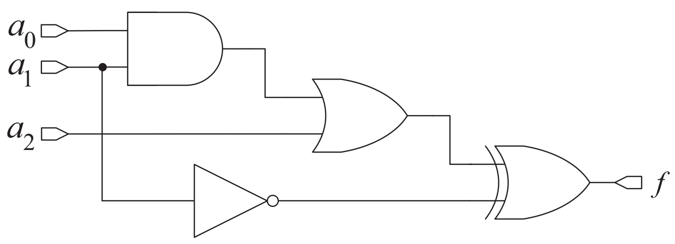


Figure 2.3: Example of gate netlist

Example 2.5. *(***Gate Netlist***) Figure 2.3 shows a gate netlist for the function $f(a_0, a_1, a_2)$ from Table 2.1.*

A Boolean network is often referred to as a *combinatorial circuit*, i.e., a circuit without any memory elements, as opposed to *sequential circuits*. In this thesis, the main focus is on combinatorial instances which can also be composed from the unrolled sequence of the combinatorial time frames of a sequential design.

2.5 Boolean Satisfiability

The *Boolean satisfiability (SAT)* problem is one of the central problems in computer science. Many important problems in logic synthesis and verification can be reduced to SAT. Due to this fact, the SAT problem was actively investigated in the research community in the last few decades.

Definition 2.18. *(***Boolean Satisfiability***) Let $f : \mathbb{B}^n \mapsto \mathbb{B}$ be a Boolean function. The Boolean satisfiability (SAT) problem is the decision problem to find such a value assignment (model) for the variables $\{x_1, \ldots, x_n\}$ that the function $f(x_1, \ldots, x_n)$ becomes satisfiable, i.e., $f(x_1, \ldots, x_n) = 1$ or prove that such an assignment does not exist, i.e., it always holds that $f(x_1, \ldots, x_n) = 0$. In the latter case, function $f(x_1, \ldots, x_n)$ is unsatisfiable.*

It is worth mentioning that equivalence checking can also be formulated as a SAT problem. Assume, it is required to check whether two formulas f_1 and f_2 are functionally equivalent, then solving the SAT problem for the expression $\phi = (f_1 \oplus f_2)$ serves the objectives of equivalence checking. For the situations when $\phi = 1$, the formulas are not equivalent under the found value assignment. This assignment is also referred to as a *counterexample*. From $\phi = 0$ for all possible assignments to the variables of f_1 and f_2, it follows that the formulas are always functionally equivalent. In practice, the function to be analyzed with SAT is represented as a CNF. The process of translating this function into the CNF is known as *bit blasting* [KS07] and can be implemented, for example, based on the *Tseitin transformations* [Tse68].

Different strategies for SAT solving are available. They can be divided into two categories, namely *incomplete* algorithms and *complete* algorithms. The first category of algorithms performs a local search for a satisfiable assignment in a function and, therefore, it is not capable to prove its unsatisfiability. On the contrary, given enough resources, complete algorithms can prove unsatisfiability of a function or find a satisfiable assignment if it exists. In this thesis, only complete algorithms are considered.

One possible way to solve a SAT problem is based on an iterative application of the resolution operation. This algorithm is also known as a *Davis-Putman* procedure [DP60]. Here, a CNF formula $f = \bigwedge_{i=0}^{k} C_i$ is taken as an input for the procedure, where C_i denotes the *i-th* clause of the formula. At each iteration for the case $n \neq m$, all clauses C_n and C_m are considered if there exists a variable a such that a appears exactly once in C_n and its complement $\overline{a}$ appears in C_m. Then, the *resolvent* is generated in the form $C_{n,m} =$

$(C_{n\setminus a} \vee C_{m\setminus\overline{a}})$, where $C_{n\setminus a}$ denotes the clause C_n without the literal a and $C_{m\setminus\overline{a}}$ stands for the clause C_m without the literal $\overline{a}$. As soon as some resolvent results in an empty clause, it means that the formula is unsatisfiable. Otherwise, if no more new resolvents can be generated for the function, it is proved that a satisfiable assignment exists for this function. In the worst case, the number of generated resolvents is exponential. Thus, in practice, this algorithm is only applicable for small formulas. The algorithm becomes infeasible for large problems with thousands of variables. Therefore, to deal with real-world problem instances, a more sophisticated algorithm is needed.

The basis for almost all complete modern SAT solvers is the so-called *Davis-Putman-Logemann-Loveland (DPLL) algorithm* that was proposed in [DLL62] and is an extension of [DP60]. The main idea behind the DPPL procedure is an enumeration of all possible assignments for the variables of the given CNF formula. In the literature, the set of all possible assignments for the given CNF formula is also denoted as *search space*. The formula is evaluated under each assignment and, by this manner, is iteratively checked for the satisfiability. Here, one value for each variable in the CNF is tried at a time. In practice, the choice of the next *decision variable* depends on a heuristic. The process of assigning a value for a decision variable is also referred to as *branching*. It is necessary to note that the variable assignments can either be chosen by means of a decision or be implied based on reasoning, due to the application of the following rules:

- *pure literal rule (PLR)*: if the variable a appears only in one form (negative or positive) throughout the CNF then the value is selected for a such that all clauses containing a become satisfied,
- *unit clause rule*: in every *unit* clause, i.e., a non-satisfied clause with exactly one unassigned literal b, the value is selected for the variable of b such that the clause becomes satisfied.

Iterative application of the unit clause rule is called *Boolean constraint propagation (BCP)*. Branching, PLR, and BCP can cause *conflicts* in a CNF. A conflict is a situation when a clause is unsatisfiable under the current assignment. In this case, the algorithm backtracks to the most recently taken decision, cancels all assignments deduced in-between, and, thereafter, tries an unenumerated value. This process is also named *chronological backtracking*.

```
DPLL($\Phi$) {
    while $true$ {
        if !$make\_branch()$
            return SAT;
        while !$bcp()$ {
            if !$backtrack()$
                return UNSAT;
        }
    }
}
```

Figure 2.4: Naive DPLL algorithm

A pseudocode for the naive DPLL algorithm is sketched in Figure 2.4. SAT solving for a CNF Φ begins with the routine *make_branch()* which is responsible to make an assignment for the next free decision variable. This routine returns *false* iff no unassigned variable is left in the CNF or, in other words, a model for Φ is found. Otherwise, *make_branch()* ends with *true* and

DPLL proceeds with PLR and BCP implemented in *bcp()*. If a conflict occurs during BCP then *bcp()* returns *false* and *backtrack()* is invoked. *Backtrack()* cancels all value assignments up to the recent decision variable, flips the value for this variable and returns *true*. However, in case when, at the top level, both values were already tried for the variable and both assignments led to a conflict in the CNF then Φ is declared unsatisfiable. As a consequence, *backtrack()* provides *false* at its output and *DPLL* terminates with *UNSAT*.

In the past, effective improvements for SAT solving have been proposed. One such advancement is *clause learning* which is invoked as soon as a conflict occurs. During conflict analysis, the implication relations, expressed by means of an *implication graph*, are analyzed to learn a conflict, and, then, a new conflict clause is generated. This clause is redundant in the sense that it does not change the function of the CNF formula but avoids repetition of the same mistake later in the search. In other words, this clause prunes the search space. *Incremental* SAT solvers add such a clause on the fly into the CNF database.

Non-chronological backtracking is an example of another efficient enhancement for SAT solving. Some conflict-learning schemes, like, e.g., the ones based on the concept of the *unique implication point* [ZMMM01], make it possible to find the reason for a conflict on earlier decision levels than the level of the current decision variable. Thus, the search space can further be reduced because larger unsatisfiable subspaces are cut off.

For the overview on the other optimizations proposed for SAT, as, for instance, *two-watched-literals scheme*, *restarts*, *clause deletion* and so on, the reader is referred to [BHvMW09].

In spite of the exponential size in the search space of a SAT instance, the improvements made for SAT solving have significantly enhanced the efficiency of the DPLL algorithm. There was developed and introduced a series of well-implemented SAT solvers, for example, *GRASP* [MSS99], *Chaff* [MMZ$^+$01], *BerkMin* [GN02], *MiniSat* [ES03], *PrecoSAT* [Pre], and others. All of them have demonstrated a good performance in practice. Unfortunately, they usually perform poorly on instances containing bit-blasted arithmetic operations derived from, e.g., data-path designs.

2.6 Satisfiability Modulo Theory

As mentioned in the previous section, a SAT solver is able to reason only at the *bit level*. Thus, if the instance in question is initially described at higher levels of abstraction then it has to be converted to CNF. In cases like this, however, the information from the higher levels of abstraction is no longer available for reasoning at the bit level. Let us illustrate this problem with the help of a small example.

Example 2.6. *(***High Level of Abstraction vs. Bit Level***) Assume the following inequation:*

$$((a+b)\cdot c) \neq (a\cdot c + b\cdot c),$$

where $a, b, c \in \mathbb{Z}_n$. Let us further assume that it is required to check whether an satisfiable value assignment exists for the inequation. At the integer level, after application of a *rewrite rule derived from the distributive law, it is easy to conclude that the inequation has no solution. In contrast to that, consuming resources, a SAT solver will enumerate all possible assignments first and, then, provides the same result.*

Such a loss of high level information as explained above is disadvantageous from the practical point of view. For example, digital designs are usually implemented at the *register transfer level (RTL)* so that some problems for hardware verification can be formulated at the *word level*, i.e., a level of abstraction where operations over bit vectors are used. At the same time, the RTL semantic defines logic operations over individual bits. This makes an RTL language more powerful, flexible and precise on the one hand and results in an instance under verification with mixed word-/bit-level functions and operators on the other hand. So, development of efficient algorithms to use the full potential of SAT solving in combination with high level information is an active research field at the moment. This problem is addressed by *Satisfiability Modulo Theory (SMT)* solvers [BSST09].

Definition 2.19. *(***SMT Problem***) An SMT problem is a decision problem for a logical formula in combination with some background theory expressed in first-order logic.*

Since *first-order logic* (also called *predicate logic*) is crucial for understanding the SMT concept, a brief definition of this logic is given below; for a deeper study, the reader may refer to standard textbooks like, e.g., [Bar77, And02, KS08].

As it is pointed out in [KS08], the following elements are basis for first-order logic:

- variables,
- logical symbols:
  - Boolean connectives like, e.g., $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$,
  - quantifiers: $\exists$, $\forall$.
- non-logical symbols: constants, predicate and function symbols,
- syntax: rules for constructing formulas.

In principle, first-order logic is a formal system for propositions with regard to variables, fixed functions and predicates. The first-order logic can be thought as an extension of propositional logic.

The symbols $\exists$ and $\forall$ are used to denote *quantifications*. To be more exact:

- $\exists$ is an *existential quantifier*, i.e., $\exists a \in A : P(a)$ (usually written as $\exists a P(a)$) means that in the set A there exists an element a such that the predicate $P(a)$ holds,
- $\forall$ is an *universal quantifier*, i.e., $\forall b \in B : P(b)$ (usually written as $\forall b P(b)$) means that the predicate $P(b)$ holds for any element b of the set B.

A set of non-logical symbols is also known as a *signature*. A *predicate* is a binary-valued function which can be defined over non-binary variables as, e.g., an inequation like $f_1 \leq f_2$, where $f_1 = (a + b)$ and $f_2 = (a * b)$ are arithmetic functions over integer variables.

A *free* variable is a variable which is not connected with a quantifier. Most logics studied in practice are quantifier free.

In first-order logic, atomic expressions are formulated as follows:

$$\begin{array}{rcl} \text{atom} & := & \text{predicate}(\text{term}_1, \ldots, \text{term}_n) \\ & | & \text{term}_1 = \text{term}_2 \end{array}$$

whereas every term is defined as

$$\begin{array}{rcl} \text{term} & := & \text{function}(\text{term}_1, \ldots, \text{term}_n) \\ & | & \text{constant} \\ & | & \text{variable} \end{array}$$

More complex expressions like a formula can be generated using non-logical symbols from a given signature and, in addition to that, using logical symbols. Thus, for a given signature Σ, a Σ-formula can be derived. A formula without free variables is called a *sentence*.

Example 2.7. *The following expression is a first-order formula:*

$$\forall a \exists b \exists c (a > 0) \wedge (b > 0) \wedge (c > 0) \wedge (a > b) \Rightarrow (a^c > b^c),$$

where $a, b, c \in \mathbb{Z}$.

With this basic knowledge about first-order logic, one may define a first-order theory, e.g., as stated in [KS08]. A *first-order Σ-theory T* consists of a set of Σ-sentences. Moreover, for a *Σ-theory T*, a *T-satisfiable formula* and a *T-valid formula* can be defined. More precisely, a Σ-formula ϕ is T-satisfiable if there exists a structure that satisfies both the formula and the sentences of the theory T. One says that a Σ-formula ϕ is T-valid if all structures that satisfy the sentences of the theory T also satisfy the formula ϕ. It should be noted that the set of sentences needed to define a theory is usually large. Therefore, for the definition of a theory, a set of axioms is used such that all the required sentences can be inferred from these axioms.

There are numerous theories compatible with SMT. They depend on the domain of application. Here, some examples are the theory of equality with uninterpreted functions, the theory of integers, the theory of arrays, the theory of bit vectors, and others. It is also important to mention that some theories describing real-world problems are undecidable per se. For instance, the dynamic systems with continuous behaviour can mathematically be expressed in terms of non-linear arithmetic over the real numbers.

2.6.1 SMT solving

Decision procedures for solving SMT formulas are more complex than for propositional logic, and these procedures can be sorted into two categories:

- the *eager* approach,
- the *lazy* approach.

The eager approach concentrates on finding an efficient technique to translate an SMT formula into an equisatisfiable CNF formula. The implementation of this approach is quite straightforward and applicable with respect to any decidable satisfiability problem of the NP class. Moreover, any SAT solver at hand can be used. Such a strategy is worthwhile in practice up to some extent only, since the sophisticated formula translations may sometimes result in an exponential blow-up and the theory-specific semantic encoded in an SMT formula is usually hidden from the reasonings in its propositional counterparts.

The lazy approach is a combination of the DPLL-based algorithm with a solver for the theory of interest, where the interaction between them is implemented by means of a well-defined interface. Here, the Boolean space of an SMT formula is explored with the help of the integrated SAT solver which provides a satisfiable value assignment, if one exists, for the propositional part of the SMT formula. The theory solver proceeds with this assignment and checks it for feasibility with regard to the conjunction of the predicates. In case the assignment does not conflict with the predicates, the SMT formula is satisfiable and the approach terminates. Otherwise, the theory solver must be able to explain the conflict and backtrack. After that, the SAT solver continues to search for an assignment compatible with the theory. If such an assignment is not found then the approach proves unsatisfiability for the SMT formula. In the literature, it is also common to denote the lazy approach as *DPLL(T)* [GHN$^+$04], where T refers to a theory in question.

Recently, a lot of effort was dedicated to the development and refinement of SMT decision procedures, e.g., [BDL98] [BDS02] [ABC$^+$02] [NO05] [BCF$^+$07] [BKO$^+$07] [GD07]. At the same time as SMT is investigated, different SMT solvers are introduced. In practice, many of them perform on quantifier-free fragments of first-order logic, e.g., *UCLID* [SLB03], *Yices* [DdM06], *CVC3* [BT07], *Z3* [MB08], *Beaver* [JLS09], *OpenSMT* [BPST10] and so on. The reader may find further information in [RT06].

In this thesis, we are interested in a particular subset of the SMT logic, namely the quantifier-free logic over the theory of fixed-sized bit vectors or *QF_BV* for short. The semantic of QF_BV is expressive enough and suited well to formulate the verification problems over the bit vector operations as, e.g., in a data-path design. Since efficient algorithms for solving such verification problems are the main scope of this work, it is necessary to allude to some QF_BV SMT solvers proposed in the last few years: *Spear* [BH08] [Spe], *Boolector* [BB09] [boo], *MathSAT* [BBC$^+$05], and *simplifyingSTP* based on revision 939 of *STP* [GD07, STP]. Here, the techniques integrated into the engines were quite successful, so that these solvers demonstrated the best performance in the QF_BV division on the SMT competitions 2007 [SMT07], 2008 [SMT08], 2009 [SMT09], and 2010 [SMT10], respectively. However, similar to SAT, solving hard

SMT formulas with arithmetic operations, like addition and multiplication, is often not feasible because of very high costs in terms of CPU time and memory resources. In Chapter 6, we will introduce a new technique based on the algorithms of computer algebra and aimed at improving the efficiency of solving hard QF_BV SMT problems. To understand the theory of fixed-sized bit vectors, a succinct explanation of it is given in Section 2.6.2. For a thorough description on the SMT language, please, refer to its standard specifications of version 1.2 [RT06] and version 2.0 [BST10]. As an example, SMT formulas for the property checking instances of (2×2) integer multipliers are depicted in Figures A.1,A.2, A.3, A.4 of Appendix A.

2.6.2 Fixed-Size Bit Vectors

As it was already stated out in Section 2.2, a bit vector $a = a[n-1:0] = (a_{n-1}, \ldots, a_0)$ is an ordered concatenation of bits. Here, the number $n > 0$ specifies the length or the *bit width* of this bit vector. In practice, if the value of n is known for all bit vectors under investigation then such bit vectors are of a *fixed size*. In contrast to that, in non-fixed size bit vectors, the value of n remains unknown or the extraction of bits at unknown position is defined. The bit $a_0 = a[0]$ is called *the least significant bit (LSB)* and $a_{n-1} = a[n-1]$ is *the most significant bit (MSB)* of the bit vector. The bit vector representation is frequently used to encode integers in base 2. For numbers it is common to distinguish between an *unsigned* and a *signed* mode. In the latter case, a *two's complement* encoding is usually applied for binary numbers. Here, the MSB is considered as a *sign bit*. Unless stated otherwise, we will assume all bit vectors to be of unsigned type.

Moreover, throughout this thesis the following notations are used:

- for $a \in \mathbb{Z}$, $b > 0$ the remainder $a \bmod b$ of the integer division a/b denotes the smallest $k \geq 0$ with $k = a - mb$ for some $m \in \mathbb{Z}$. Often, one says that k is equal to *a modulo b*,

- the unsigned integer represented with a bit vector $a = (a_{n-1}, \ldots, a_0)$ is denoted by $\mathbb{Z}^+(a) = \sum_{i=0}^{n-1} 2^i a_i$, where 2^i is called a *weight* of the *i-th* bit and $\mathbb{Z}^+(a)$ is a *weighted sum* of a,

- alternatively, the signed integer represented with a bit vector $a = (a_{n-1}, \ldots, a_0)$ is denoted by $\mathbb{Z}(a) = (-2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i)$. This is a so-called *two's complement* conversion rule,

- conversely, $\langle x, n \rangle$ for $n > 0$ and $x \in \mathbb{Z}$ denotes the uniquely determined bit vector $a = (a_{n-1}, \ldots, a_0)$ with $(x \bmod 2^n) = \sum_{i=0}^{n-1} 2^i a_i$, i.e., a is the n-bit binary unsigned integer representation of x.

Thus, bit vectors are a proper way to model data storage in the digital designs of a data path and memory. Since any data path must be able to *process* data, operations over bit vectors have to be defined. A list of typical bit-vector operations is shown in Table 2.4. The detailed semantics on them can be found in [RT06]. Here, depending on

Name	**Syntax**
	Unary
Bitwise negation	bvnot $a[n:0]$ $r[n:0]$
Two's complement minus	bvneg $a[n:0]$ $r[n:0]$
	Binary
Bitwise OR	bvor $a[n:0]$ $b[n:0]$ $r[n:0]$
Bitwise AND	bvand $a[n:0]$ $b[n:0]$ $r[n:0]$
Bitwise XOR	bvxor $a[n:0]$ $b[n:0]$ $r[n:0]$
Addition modulo 2^n	bvadd $a[n:0]$ $b[n:0]$ $r[n:0]$
Two's complement subtraction	bvsub $a[n:0]$ $b[n:0]$ $r[n:0]$
Multiplication modulo 2^n	bvmul $a[n:0]$ $b[n:0]$ $r[n:0]$
Division, undefined if $\mathbb{Z}(b) = 0$	bvudiv $a[n:0]$ $b[n:0]$ $r[n:0]$
Remainder, undefined if $\mathbb{Z}(b) = 0$	bvurem $a[n:0]$ $b[n:0]$ $r[n:0]$
Predicate less than	bvult $a[n:0]$ $b[n:0]$ $r[0:0]$
Predicate equal to	$=$ $a[n:0]$ $b[n:0]$ $r[0:0]$
Bits concatenation	concat $a[n:0]$ $b[m:0]$ $r[(n+m+1):0]$
Bits slicing, where $i, j \in \mathbb{Z}^+$	extract i j $a[m:0]$ $r[(i-j):0]$
Shift left	bvshl $a[n:0]$ $b[n:0]$ $r[n:0]$
Logical shift right	bvlshr $a[n:0]$ $b[n:0]$ $r[n:0]$
	Ternary
If then else	ite $a[0:0]$ $b[n:0]$ $c[n:0]$ $r[n:0]$

Table 2.4: Bit-vector operations

the number of the input arguments, all bit-vector functions considered in this thesis can be classified into three groups: *unary*, *binary*, and *ternary*. For each of these functions the bit vector r denotes the *output argument* whereas a, b, and c are the notations for the first, the second, and the third *input argument*, respectively. All these operations serve as a good basis in the practice to implement more complex and sophisticated operations on demand. The importance of the theory of the bit vectors in a computer architecture design is well described in [HP02]. The author of [Kor98] provides a good description on computer arithmetic algorithms over bit vectors.

2.7 Basics of Computer Algebra

This section seeks to familiarize the reader with some basics of computer algebra. Here, the main topic of interest is the theory of *Gröbner bases*, also known as *standard bases*, as the understanding of this theory is necessary to proceed with the new verification approach described in Chapter 4 where algebraic techniques are applied.

For the sake of simplicity and a proper introduction to this theory, especially for readers without any background in computer algebra, this section is split into three parts as follows:

- Section 2.7.1 contains definitions for the central objects of computer algebra: *commutative ring*, *field*, *polynomial*, *variety*, *ideal*, *base*,
- in Section 2.7.2 according to the notions of [CLO07] by considering a particular case for a polynomial ring $k[x_1, \ldots, x_n]$ over a field k, a typical interpretation for the concept of Gröbner bases is given,
- in Section 2.7.3, due to the reason that the approach of Chapter 4 is applicable not to a field but to a ring only, as determined by [BDG$^+$09], the theory of Gröbner bases is further generalized for the case of a ring R.

Indeed, the scope of this section is limited to the main notions and the underlying algorithms, thus, for a deeper study of the theory the reader may refer to textbooks like, e.g., [AL03, CLO07, GP07].

2.7.1 Preliminaries

Definition 2.20. *(***Commutative Ring***) A commutative ring is a set R equipped with two binary operations, namely addition ($+$) and multiplication ($\cdot$), defined on R such that the following laws hold:*

- *commutativity: $a + b = b + a$ and $a \cdot b = b \cdot a$ for all $a, b \in R$,*
- *associativity: $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in R$,*
- *distributivity: $a \cdot b + a \cdot c = a \cdot (b + c)$ for all $a, b, c \in R$,*

- *identities: there exist elements* $0, 1 \in R$ *so that* $a + 0 = a \cdot 1 = a$ *for all* $a \in R$,
- *additive inverses: given* $a \in R$, *there exists* $b \in R$ *so that* $a + b = 0$.

A common example of a commutative ring is the set of integers $\mathbb{Z}$ where the ordinary operations of addition, subtraction and multiplication can be applied.

Note that the existence of the element 1 is usually not required. However, in this thesis, we only consider rings with 1.

Definition 2.21. *(***Field***) A field is a commutative ring k with multiplicative inverses, i.e., for each $a \in k$, where $a \neq 0$, there exists $c \in k$ such that $a \cdot c = 1$.*

It is obvious that in addition to the operations of a commutative ring, a field defines the operation of division. Note that any field is a ring, but the opposite is not true.

Example 2.8. *(***Field***) The set of all real numbers $\mathbb{R}$ is a field whereas the set of all integers $\mathbb{Z}$ is not a field as, e.g.,* $(5/3) \notin \mathbb{Z}$.

Before the concept of a *polynomial* will be explained, the notion of a *monomial* has to be introduced [1].

Definition 2.22. *(***Monomial***) A monomial in variables $x_1, x_2, \ldots, x_n$ is a power product $x^\alpha = x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdot \ldots x_n^{\alpha_n}$, where $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in \mathbb{N}^n$. The measure $|\alpha| = \alpha_1 + \alpha_2 + \ldots, \alpha_n$ is called the total degree of the monomial.*

Definition 2.23. *(***Polynomial***) Given a ring R and the variables $x_1, x_2, \ldots, x_n$, a polynomial f is a finite linear combination of monomials in the following form:*

$$f = \sum_\alpha a_\alpha x^\alpha, \ a_\alpha \in R \backslash \{0\},$$

where $a_\alpha x^\alpha$ is called a term of the polynomial and a_α is the coefficient of the term. Here, the maximum $|\alpha|$ among all terms in f is the total degree of f and denoted by deg(f).

A polynomial in one variable is known as a *univariate polynomial*, and a polynomial in more than one variable is a *multivariate polynomial*.

For a ring R, the set of all possible polynomials in the variables $x_1, x_2, \ldots, x_n$ is denoted by $R[x_1, x_2, \ldots, x_n]$. As stated in [GP07], with the usual addition and multiplication, see Formula 2.1 and Formula 2.2, respectively, the structure $R[x_1, x_2, \ldots, x_n]$ is again a ring, called a *polynomial ring* in n variables over R. Moreover, it is easy to prove that this structure fulfills, indeed, the conditions of a commutative ring.

$$\sum_\alpha a_\alpha x^\alpha + \sum_\alpha b_\alpha x^\alpha := \sum_\alpha (a_\alpha + b_\alpha)^\alpha \tag{2.1}$$

[1]One should not confuse a monomial used in computer algebra with a Boolean monomial defined in Section 2.4.3

$$(\sum_{\alpha} a_\alpha x^\alpha) \cdot (\sum_{\beta} b_\beta x^\beta) := \sum_{\gamma} (\sum_{\alpha+\beta=\gamma} a_\alpha b_\beta) x^\gamma \tag{2.2}$$

Similar to that for the case of a field k, the set of all possible polynomials in the variables $x_1, x_2, \dots, x_n$ is indicated with $k[x_1, x_2, \dots, x_n]$. Consequently, $k[x_1, x_2, \dots, x_n]$ also fulfills the conditions of a commutative ring. Note here that, the multiplicative inverses do not exist in general, e.g., $\frac{1}{x_1} \notin R[x_1, x_2, \dots, x_n]$. Hence, $R[x_1, x_2, \dots, x_n]$ is no longer a field and, therefore, named polynomial *ring*.

Often in computer algebra, it is necessary to have a monomial ordering so that the terms are sorted inside a polynomial f, e.g., $f = a_{\alpha_1} x^{\alpha_1} + a_{\alpha_2} x^{\alpha_2} + \dots a_{\alpha_m} x^{\alpha_m}$, and some special elements of f can be identified as follows:

- $LT(f) = a_{\alpha_1} x^{\alpha_1}$: the leading term of f with $a_{\alpha_1} \neq 0$,
- $LM(f) = x^{\alpha_1}$: the leading monomial of f,
- $LC(f) = a_{\alpha_1}$: the leading coefficient of f,
- multideg(f) = α_1: multidegree of f.

Usually, a *global ordering* $<$ on monomials is required. In fact, monomial orderings must be preserved under multiplication. Additionally, in global orderings for each ring variable v_i, the condition $v_i > 1$ must always be fulfilled.

Example 2.9. *(***Monomial Ordering***) Let us consider two multivariate polynomials* $p_1 = 10xy^4 + 15xz^9 + 7x^5$ *and* $p_2 = 3x^5 + 5xz^9$*. With regard to the* lexicographical order (lex-order)*, the polynomial* $p = p_1 + p_2$ *has to be written as* $p = 10x^5 + 10xy^4 + 20xz^9$*, with* $LT(p) = 10x^5$*,* $LM(p) = x^5$*,* $LC(p) = 10$*, and multideg(p) = (*$5, 0, 0$*).*

A more detailed description of monomial orderings and the examples on that can be found in, e.g., [AL03, CLO07, GP07].

Any polynomial f may be transformed into a polynomial equation, i.e., an expression of the form $f = 0$. In practice, it is common to solve a system of polynomial equations. The set of all solutions for such a system is called *(algebraic) variety* and usually denoted by V.

Definition 2.24. *(***Variety***) For a ring R and the polynomials $f_1, \dots, f_m \in R[x_1, \dots, x_n]$, the variety $V(f_1, \dots, f_m)$ is defined as follows:*

$$V(f_1, \dots, f_m) = \{(a_1, \dots a_n) \in R^n : f_i(a_1, \dots a_n) = 0,\ 1 \leq i \leq m\},$$

where $R^n = \{(a_1, \dots, a_n) : a_1, \dots, a_n \in R\}$ is the n-dimensional affine space over R.

Moreover, any field k, where every non-constant polynomial $p \in k$ has all its roots, i.e., all solutions for $p = 0$ in k, is called an *algebraically closed field*.

While considering a ring R, special subsets of R, known as *ideals*, can be identified.

Definition 2.25. (**Ideal**) *Given a ring R and a subset $I \subset R$, I is an ideal if the following conditions hold:*

- $0 \in I$,
- $f + g \in I$ *for any* $f, g \in I$,
- $f \cdot g \in I$ *for any* $f \in I$ *and any* $g \in R$.

Since a polynomial ring $R[x_1, \dots, x_n]$ is considered in Chapter 4 and Chapter 5 of this thesis, a way to generate an ideal for such a ring is explained with the next lemma.

Lemma 2.1. *Given polynomials $f_1, f_2, \dots, f_s \in R[x_1, x_2, \dots x_n]$, the set*

$$\langle f_1, f_2, \dots, f_s \rangle := \left\{ \sum_{i=1}^{s} h_i f_i : h_1, \dots, h_s \in R[x_1, x_2, \dots x_n] \right\}$$

is an ideal in $R[x_1, x_2, \dots x_n]$. We say that this ideal is generated by $f_1, f_2, \dots, f_s$.
Proof: *see Definition 2 and Lemma 3 in §4 of [CLO07] by taking into account the fact that the ring $k[x_1, x_2, \dots x_n]$ is a special case of the ring $R[x_1, x_2, \dots x_n]$.* □

For the ideal $I = \langle f_1, f_2, \dots, f_s \rangle$, the set of generating polynomials $\{f_1, f_2, \dots, f_s\}$ forms a sort of *basis* of the ideal. Indeed, from a set of equations $f_1 = 0, f_2 = 0, \dots, f_s = 0$ with $f_i \in R[x_1, x_2, \dots x_n]$ for $1 \leq i \leq s$, one may formulate:

$$h_1 f_1 + h_2 f_2 + \dots + h_s f_s = 0, \tag{2.3}$$

where $h_i \in R[x_1, x_2, \dots x_n]$. It is evident that the left-hand side of Formula 2.3 belongs to $I = \langle f_1, f_2, \dots, f_s \rangle$, i.e., I can be considered as a set of all "polynomial sequences" from $f_1 = f_2 = \dots = f_s = 0$.

Based on the leading term of a polynomial, we can define the leading term ideal.

Definition 2.26. *Let $I \subset R[x_1, x_2, \dots x_n]$ be an ideal such that $I \neq \{0\}$. Then:*

- $LT(I)$ *indicates the set of all leading terms of non-zero elements in I, where:*

$$LT(I) = \{cx^\alpha : \exists f \in I, LT(f) = cx^\alpha\},$$

- $\langle LT(I) \rangle$ *stands for the ideal generated from the elements of* $LT(I)$.

Among other objects, an ideal is a very important algebraic structure in computer algebra. Many mathematical problems can be formulated as a so-called *ideal membership problem*, i.e., to check whether a polynomial f belongs to some ideal $I = \langle f_1, \dots, f_s \rangle$ generated from a set of polynomials $\{f_1, \dots, f_s\}$. This problem can efficiently be solved by calculating the normal form of the polynomial f with respect to a so-called Gröbner basis G of the ideal I. Section 2.7.2 continues with the discussion on ideals and describes the background of a methodology towards solving the ideal membership problem with regard to a field k. Further, in Section 2.7.3, the idea of the methodology is explained for a ring R.

2.7.2 Standard Bases over a Field

In this section we continue studying ideals and explain the concept for solving the ideal membership problem over a field k. At first, we extend the above notion of varieties to varieties of ideals.

For any nonempty ideal with even infinitely many polynomials, the set of all roots ($V(I)$) in this ideal can be determined by a finite set of polynomial equations. This is stated by the following proposition and subsequent remarks.

Proposition 2.1. *Given an ideal $I \subset R[x_1, \dots, x_n]$ with $I = \langle f_1, \dots, f_s \rangle$ and the algebraic variety $V(I)$ defined as*

$$V(I) = \{(a_1, \dots, a_n) \in R^n : f(a_1, \dots, a_n) = 0, \forall f \in I\},$$

then

$$V(I) = V(f_1, \dots, f_s).$$

Proof: *see Proposition 9 in §5 of Chapter 2 in [CLO07].* □

According to the *Hilbert's Nullstellensatz*, see Theorems 1, 2 of §1 in Chapter 4 of [CLO07], for any algebraically closed field $k[x_1, \dots, x_n]$, it is true that $V(I) = \emptyset$ if and only if $I = k[x_1, \dots, x_n]$. Thus, to ensure the existence of a nonempty variety for the ideal I in the polynomial ring k, it is enough to prove that:

- I is not the ring itself,
- the ring is algebraically closed.

On the other hand, as follows from *Hilbert Basis Theorem*, see, e.g., Chapter 2 of [CLO07], any ideal in $k[x_1, \dots x_n]$ is finitely generated, i.e., for each ideal I there is a generator set $f_1, \dots, f_s$ such that $I = \langle f_1, \dots, f_s \rangle$. Moreover, there may exist different generator sets for I. One generator with specific properties, called *Gröbner basis*, can be generated for any nonempty ideal I.

Before the concept of Gröbner basis will be defined, it is necessary to introduce the *division algorithm* on polynomials. Among all other operations over polynomials, the importance of this algorithm is crucial in computer algebra.

Theorem 2.1. *(***Division Algorithm***) Given a polynomial ring $k[x_1, \dots, x_n]$ and a global monomial ordering $<$, for any ordered polynomials $f_1, \dots, f_s \in k[x_1, \dots, x_n]$, every $f \in k[x_1, \dots, x_n]$ can be expressed in the following form:*

$$f = a_1 f_1 + \cdots + a_s f_s + r,$$

where $a_i, r \in k[x_1, \dots, x_n]$. Moreover, either $r = 0$ or none of $LT(f_1), \dots LT(f_s)$ divides r. Besides that, it always holds that multideg(f) $\geq$ multideg($a_i f_i$) for any $a_i f_i \neq 0$, where $\neq$ denotes the natural partial order on $\mathbb{N}$.

Proof: *see Theorem 3 in §3 of Chapter 2 in [CLO07]. The pseudocode of the division algorithm is depicted in Figure 2.5.* □

Example 2.10. *(***Division***) Assume the lex-ordered polynomials $f = x^3 - x$, $f_1 = x^2 - 1$, and $f_2 = x - y$, where $f, f_1, f_2 \in k[x_1, x_2]$. It is requested to divide f by f_1 and f_2. After all the iterations, the division algorithm results in $a_1 = x$, $a_2 = 0$, and $r = 0$, thus, one may write $f = x^3 - x = x(x^2 - 1) + 0(x - y) + 0$.*

```
DivisionAlgorithm($f, f_1, \dots f_s$) {
    $a_1 := 0, \dots, a_s := 0$;
    $r := 0$;
    $p := f$;
    while $p \neq 0$ {
        $i := 1$;
        $is\_division := false$;
        while $(i \leq s$ & $!is\_division)$ {
            if $LT(f_i)$ divides $LT(p)$ {
                $a_i := a_i + LT(p)/LT(f_i)$;
                $p := p - (LT(p)/LT(f_i))f_i$;
                $is\_division := true$;
            }
            else
                $i := i + 1$;
        }
        if $!is\_division$ {
            $r := r + LT(p)$;
            $p := p - LT(p)$;
        }
    }
    return $a_1, \dots, a_s, r$;
}
```

Figure 2.5: Division algorithm in a polynomial ring $k[x_1, \dots, x_n]$

The division algorithm of Theorem 2.1 provides a very nice feature, namely it can be applied to solve the ideal membership problem. As proved in Chapter 2 of [CLO07] for an ideal $I = \langle f_1, \dots, f_s \rangle$, if after division f by $f_1, \dots, f_s$ the remainder r becomes zero then f is a member of the ideal I. However, the converse is in general not true.

Example 2.11. *(***Division, continued***) Let us again consider Example 2.10. Assume now, the polynomials $f_1, f_2 \in k[x_1, x_2]$ form the ideal $I = \langle f_1, f_2 \rangle$. According to Chapter 2 of [CLO07], f is a part of $I = \langle f_1, f_2 \rangle$ due to the fact that $r = 0$ after division of f by f_1, f_2. However, surprisingly, if the divisors are swapped, i.e., $f_1 = x - y$ and $f_2 = x^2 - 1$, then a nonzero remainder is obtained. This time, the division algorithm* *yields $a_1 = x^2 + xy + y^2 - 1$, $a_2 = 0$, and $r = y^3 - y$, so that $f = x^3 - x = (x^2 + xy + y^2 - 1)(x - y) + 0(x^2 - 1) + y^3 - y$. Thus, it becomes obvious that the condition $r = 0$ is sufficient but not necessary to solve the ideal membership problem.*

As already mentioned above in this section, there may exist different generator sets for one and the same ideal. Then, it becomes natural to ask for a set $g_1, \dots, g_n$ so that for the case when $f \in I = \langle g_1, \dots, g_n \rangle$, the condition $r = 0$ is always implied, regardless of the ordering of the g_i. Fortunately, such a set exists for any nonzero ideal. It is known as a *Gröbner basis*.

Definition 2.27. *(***Gröbner Basis***) Given a polynomial ring $k[x_1, \dots, x_n]$, a global monomial ordering $<$ and an ideal $I \in k[x_1, \dots, x_n]$, the finite subset $G = \{g_1, \dots, g_s\} \subset I$ is a Gröbner basis (also referred to as a standard basis) if*

$$\langle LT(I) \rangle = \langle LT(g_1), \dots, LT(g_s) \rangle.$$

Proposition 2.2. *For a given monomial order, there exist a Gröbner basis G for any nonempty ideal $I \subset k[x_1, \dots, x_n]$. Any Gröbner basis of the ideal I generates I.*
Proof: *see §5 of Chapter 2 in [CLO07].* □

Proposition 2.3. *Given a Gröbner basis $G = \{g_1, \dots, g_s\}$ for the ideal $I \subset k[x_1, \dots, x_n]$. Then $f \in I$ iff the remainder r becomes zero when dividing f by G.*
Proof: *see §6 of Chapter 2 in [CLO07].* □

The remainder r after the division of f by a Gröbner basis G is also known as a *normal form* of f.

Proposition 2.4. *Given a polynomial $f \in k[x_1, \dots, x_n]$ and a Gröbner basis $G = \{g_1, \dots, g_s\}$ for the ideal $I \subset k[x_1, \dots, x_n]$. There exists a unique $r \in k[x_1, \dots, x_n]$ so that:*

- *$f = g + r$ for some $g \in I$,*
- *no term of r is divisible by any of $LT(g_1), \dots, LT(g_s)$.*

In other words, r is the unique remainder of f with respect to division by G.
Proof: *see Proposition 1 in §6 of Chapter 2 in [CLO07].* □

Obviously, since a Gröbner basis has these nice properties, such a basis is especially useful in practical applications. Therefore, a method to compute this basis is needed. In [Buc76] *Buchberger* has proved that a set of generators of any nonempty ideal can be transformed into a Gröbner basis. To enable this transformation, a special algorithm, known as the *Buchberger algorithm*, was developed. A simplified version of this algorithm is depicted in Figure 2.6. Here, the notation $S(p, q)$ means an S-polynomial introduced by Definition 2.28 and the notation $\overline{S(p, q)}^{G'}$ stands for the remainder after the division of $S(p, q)$ by G'.

Definition 2.28. *(***S-Polynomial***) Given nonzero polynomials $f, g \in k[x_1, \ldots, x_n]$, then the S-polynomial is defined as follows:*

$$S(f,g) = \frac{x^\gamma}{LT(f)} \cdot f - \frac{x^\gamma}{LT(g)} \cdot g,$$

where x^γ is the least common multiple of $LM(f)$ and $LM(g)$, i.e., $\gamma = (\gamma_1, \ldots, \gamma_n)$ with $\gamma_i = max(\alpha_i, \beta_i)$, where $\alpha = multideg(f)$ and $\beta = multideg(g)$.

```
1: GröbnerBasis($F, G$) {
2:    $G := F$;
3:    do {
4:       $G' := G$;
5:       for each pair $(p, q)$, where $p \neq q \in G'$ {
6:          $S := \overline{S(p,q)}^{G'}$;
7:          if $S \neq 0$
8:             $G := G \cup \{S\}$;
9:       }
10:   }
11:   while $G \neq G'$;
12:   return $G$;
13: }
```

Figure 2.6: Buchberger's algorithm to generate a Gröbner basis $G = \{g_1, \ldots g_t\}$ for an ideal $I = \langle f_1, \ldots, f_s \rangle$ generated by a set of polynomials $F = \{f_1, \ldots f_s\}$

Furthermore, it is also possible to compute the *reduced unique Gröbner basis*.

Definition 2.29. *(***Reduced Gröbner Basis***) Given an ideal I, the reduced Gröbner basis G of I is a Gröbner basis such that:*

- $LC(p) = 1$ *for each* $p \in G$,
- *no monomial of the polynomial* $p \in G$ *belongs to* $\langle LT(G - \{p\}) \rangle$.

Thereby, any two ideals can easily be tested for equality. In this case, one computes the reduced Gröbner bases for both ideals (with respect to the given monomial order). If the bases are identical then the ideals are equal.

2.7.3 Standard Bases over a Ring

The theory of Gröbner bases over a field k, reviewed in Section 2.7.2, is in this section extended for the case of a polynomial ring $R[x_1, \ldots, x_n]$, where R stands for a commutative ring with constant 1. Moreover, while considering polynomials, a global monomial ordering $<$ shall always be assumed.

Indeed, in this thesis we are interested in polynomials defined over the ring of integers $\mathbb{Z}$ modulo 2^n, denoted as $\mathbb{Z}/2^n[x_1, \ldots, x_n]$. Such polynomials are suited well to mathematically describe an arithmetic part of a data-path design. Thus, a decision problem to prove correctness of the arithmetic part can be represented and solved by means of a system of polynomial equations, as shown in Chapter 4.

Here, for a polynomial f with respect to a Gröbner basis G, a calculation of a normal form NF($f \mid G$) is crucial. A Gröbner basis may be generated in the same manner as explained in Section 2.7.2. However, in contrast to a field k, where the normal form is the remainder of division of f by G, a special approach is needed for the case of a ring $R[x_1, \ldots, x_n]$. Therefore, for a polynomial $f \in R$, according to [BDG+09], let us give the definitions of a *t-representation* and a *standard representation* first and, afterwards, a *normal form* will be defined.

Definition 2.30. *(***t-Representation***) Given a ring $R[x_1, \ldots, x_n]$, a monomial t and some elements $f, g_1, \ldots, g_m, h_1, \ldots, h_m \in R$ so that*

$$f = \sum_{i=1}^{m} h_i g_i.$$

The above sum is a t-representation of f w.r.t. $g_1, \ldots, g_m$ if $LM(h_i g_i) \leq t$ for all i with $h_i g_i \neq 0$.

A polynomial f is in its standard representation w.r.t. $\{g_1, \ldots, g_m\}$, if it is a $LM(f)$-representation.

Definition 2.31. *(***Normal Form***) For a ring $R[x_1, \ldots, x_n]$, let $\wp$ be the set of all finite subsets G of R. Then a map*

$$NF : R \times \wp \rightarrow R, (f, G) \mapsto NF(f \mid G)$$

is called a normal form on R, if for all $G \in \wp$ the following conditions hold:

- *$NF(0 \mid G) = 0$,*
- *$NF(f \mid G) \neq 0 \Rightarrow LT(NF(f \mid G)) \notin \langle LT(G) \rangle$ for all $f \in R$,*
- *$r := f - NF(f \mid G)$ has a standard representation w.r.t. G.*

As already mentioned, a ring R is an algebraically different structure than a field k. For this reason, the calculation of a normal form in R has to be treated differently than explained in Section 2.7.2. Figure 2.7 depicts the pseudocode for the computation of a normal form for a polynomial f in the polynomial ring $R[x_1, \ldots, x_n]$.

In fact, the theory of Gröbner bases and the computations of normal forms in particular are widely applied in practice. In mathematics, a great number of various important problems can be formulated and solved in terms of a system of polynomial equations, see, e.g., [CLO07, GP07].

```
1: CalculateNormalForm($f, G$) {
2:     while ($f \neq 0$) & ($LT(f) \in \langle LT(G) \rangle$) {
3:         $J := \{j \in \{1, \dots, s\} : LM(g_j) \mid LM(f)\}$;
4:         solve $LC(f) = \sum_{j \in J} c_j \cdot LC(g_j)$ where $c_j \in C$;
5:         $f := f - \sum_{j \in J} c_j \cdot g_j \cdot \frac{LM(f)}{LM(g_j)}$;
6:     }
7:     return $f$;
8: }
```

Figure 2.7: Algorithm for a normal form calculation of a polynomial f in a ring $R[x_1, \dots, x_n]$ with a Gröbner basis $G = \{g_1, \dots, g_s\}$ and a global monomial ordering

Also in the domain of formal verification, different techniques [SKEG05, SKE07, WHAH07, SKME08, WWS+08, BDG+09] based on symbolic computations in computer algebra have been proposed.

In this thesis, Chapter 4 provides a detailed explanation of a formal approach to solve an arithmetic decision problem which is quite typical in practice when it is necessary to ensure the correctness of an arithmetic data-path design. Here, the arithmetic parts of both a data path and the property in question are described by means of a set of polynomials F over the ring $R = \mathbb{Z}[x_1, \dots, x_m]/2^n$ with $x_i \in \mathbb{B}$, where $1 \leq i \leq m$. For the proof goal, i.e., the function which compares the design against the property, see Section 3.1.3, an additional polynomial $g \in R$ is derived. As proved in [WWS+08, Wie], the set F can directly be constructed as a Gröbner basis for the ideal $I = \langle \{F\} \rangle$. From Proposition 2.1 it follows that the equation $V(I) = V(F)$ is always valid. Therefore, as shown in Chapter 4, if the *variety subset problem* $V(F) \subset V(g)$ can be solved for all possible roots in G and g, then the property holds. Actually, the statement $V(F) \subset V(g)$ is true if and only if the condition NF$(g \mid F) = 0$ is valid. Chapter 4 provides an example for this approach.

Chapter 3

Arithmetic Bit Level Verification

In Chapter 2 we studied different levels of abstraction suitable for circuit representation in digital design. In particular, we considered the *bit level*, the *word level*, and the *register transfer level (RTL)*. Depending on the level of abstraction used for the design description, appropriate techniques need to be applied to verify this design against its *specification*. Here, standard verification algorithms are usually based on BDDs, SAT solving, and their modifications. While considering combinatorial models of circuits, the proposed techniques lack performance and robustness as soon as arithmetic functions, especially multiplication, come into play. This fact has motivated the research community to seek for alternative and more expressive representations of arithmetic circuits such that formal verification becomes feasible.

This chapter consists of Section 3.1 reviewing the *arithmetic bit level (ABL)*, proposed in [SK01] for the first time and further actively studied in [WSK05], and of Section 3.2 describing the concepts of ABL-based formal approaches that have shown to be viable and very efficient for proving correctness in data-path designs. In essence, this chapter serves as a background for Chapter 4 where the pragmatic approach of *ABL normalization*, reviewed in Section 3.2.2, is replaced by a computer algebra technique with a clean mathematical foundation. However, also this technique exploits ABL information.

3.1 Arithmetic Bit Level

To understand the concept of an *ABL description* and the main principle of all ABL-based techniques, first of all, the *arithmetic bit level* has to be introduced.

Definition 3.1. *(***ABL***) An arithmetic bit level is a level of abstraction at which a combinatorial model of an arithmetic circuit can be described in terms of three components:*

- *partial products,*
- *addition networks,*
- *comparators.*

The ABL components of Definition 3.1 form a minimal set of objects required to compactly formulate and efficiently solve such a decision problem where the absence of errors in a combinatorial arithmetic circuit has to be proved. By referring to [WSBK07], let us give a formal definition for each of the ABL components first and, then, define the *ABL description* in the sequel.

3.1.1 Partial Products

A partial product bit is a result of a bitwise multiplication. In fact, a partial product bit might be computed as the Boolean conjunction over separate bits. This can be well implemented by means of *AND*-gates defined in Table 2.2. Usually, a set of partial product bits is generated with a special unit called a *partial product generator* as, for example, in the (2×2) integer multiplier depicted in Figure 3.1.

Definition 3.2. (**Partial Product Generator**) *A partial product generator P is a triple of bit vectors* (a, b, p)*, where* a *and* b *are the input operands and* p *is the output operand such that every partial product bit is defined as* $p_i = a_i \cdot b_i$.

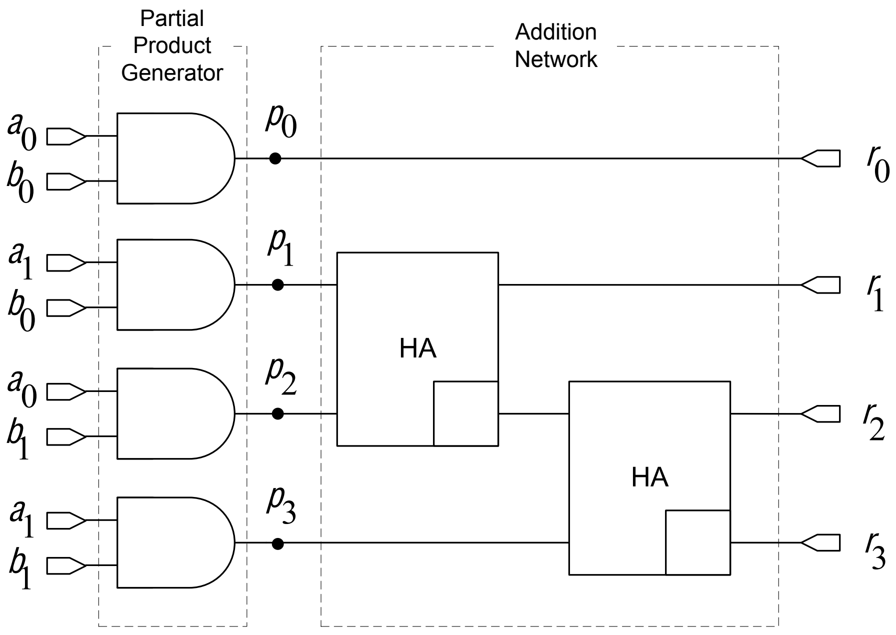
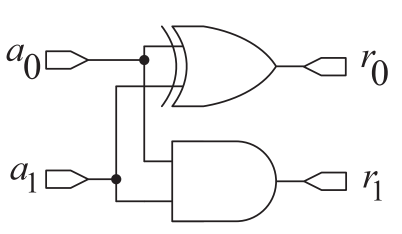


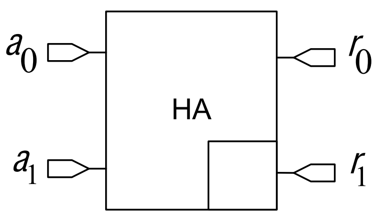
Figure 3.1: (2×2) unsigned integer multiplier at ABL

3.1.2 Addition Networks

Similar to decimal numbers, the addition of bit vectors always has to be performed with regard to weights of the individual bits. Here, the carry bits generated in column i with a weight $\hat{w}_i$ have to participate in the calculation of the next column $(i + 1)$ with the weight



(a) HA at gate level



(b) HA at ABL

Figure 3.2: Half adder (HA)

$\hat{w}_{i+1} = 2\hat{w}_i$. In other words, the addition over bit vectors results in a weighted sum over a set of Boolean variables. To enable such a bit-vector addition with the appropriate manipulation of weighted bits at ABL, an *addition network* was proposed.

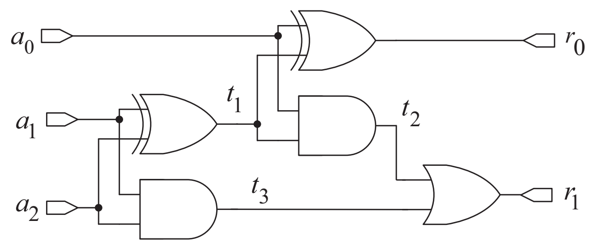
Definition 3.3. *(***Addition Network***) An addition network N is a quadruple* (A, w, c, r) *such that*

- $A = A_0 \cup A_1 \cup \cdots \cup A_n$ *is a set of input bits called addends, where every addend* $a_j \in A$ *is a Boolean variable. Besides that, for any* k *with* $0 \leq k \leq n$*, each subset* A_k *contains all addends for the column* k*,*
- $w = (w_0, \ldots, w_n)$ *is a vector of weights mapped to the addends of every column in the addition network, i.e.,* $w_k(a_m) \in \mathbb{Z}$ *and* $\forall a_m \in A_k$*,*
- $c = (c_0, \ldots, c_n)$ *is a vector of constant offsets for every column* k *with* $c_k \in \mathbb{Z}$*,*
- $r = (r_0, \ldots, r_n)$ *is a vector of Boolean variables that stand for the result bits of the addition network.*

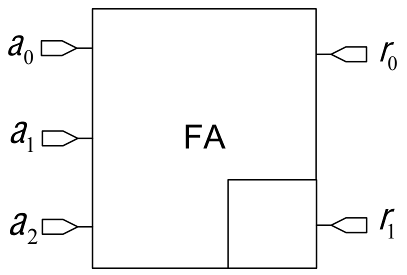
Generalizing Definition 3.3, an addition network can mathematically be described in terms of Equation 3.1

$$\mathbb{Z}(r) = (\sum_{i=0}^{n} 2^i (\sum_{a_j \in A_i} w_i(a_j) \cdot a_j + c_i)) \bmod 2^{n+1}. \tag{3.1}$$

Example 3.1. *(***Addition Network***) The most simple representatives of addition networks at ABL are a half adder (HA) and a full adder (FA). Symbols for them as well as their gate-level representations are depicted in Figure 3.2 and in Figure 3.3. It is apparent that these adders are two-column addition networks. For instance, for the FA in Figure 3.3 it holds that* $A = \{a_0, a_1, a_2\}$ *with* $w(a_i) = 2^0$ *and* $\forall a_i \in A$*,* $w = (2^0, 2^1)$*,* $c = (0, 0)$*,* $r = (r_0, r_1)$*. Often,* r_0 *is also called a sum bit and* r_1 *is referred to as a carry bit. Both HAs and FAs are widely used in practice as fundamental building blocks of a bit-vector-addition circuit. Combinations of these primitive ABL units interconnected in a proper way make it possible to model more complex addition schemes for the bit vector*



(a) FA at gate level



(b) FA at ABL

Figure 3.3: Full adder (FA)

arithmetic. As a didactic example, Figure 3.1 demonstrates the addition network over the partial product bits in a $(2{\times}2)$ *unsigned multiplier. Here, the chain of two HAs forms a so-called ripple-carry adder.*

Note that addition network may have multiple implementations by a netlist of FA/HA like, e.g., *carry-save adder (CSA)*, *carry-lookahead adder (CLA)*, *carry-select adder*; see [HP02] for more details.

In Section 3.2.2 it will be shown how structurally dissimilar but functionally equivalent arithmetic circuits can be reduced at the ABL to a common normal form.

It is also important to mention that the addition is an underlying operation of integer arithmetic. Computer arithmetic algorithms, as for example presented in [Kor98], employ addition in designing other operations. Thus, in a digital design, arithmetic calculations are always based on additions. The execution of such calculations is usually implemented inside an *arithmetic logic unit (ALU)* which is a substantial part of the *central processor unit (CPU)* in any computer.

3.1.3 Comparators

In formal verification, a typical task is to prove correctness of a design with regards to some specification or property. In the context of arithmetic circuits, it means to check the design and its specification for functional equivalence. For purposes like this, a *comparator* is used and, at ABL, it can be defined as follows.

Definition 3.4. *(***Comparator***) A comparator C is a quadruple* $(c1, c2, o, f)$ *such that:*

- $c1 = (c1_0, \ldots, c1_n)$ *and* $c2 = (c2_0, \ldots, c2_n)$ *are bit vectors under comparison; they are called input operands,*
- $f : \mathbb{B}^{n+1} \times \mathbb{B}^{n+1} \to \mathbb{B}$ *is a comparison function,*
- o *is a Boolean variable denoting the result of the comparison.*

On the whole, the comparator is a unit that may provide the result for any operation of comparison: $<, >, \leq, \geq, \neq$ and $=$. However, in the following we only consider the case of equivalence, i.e., when

$$o = f(c1, c2) = \begin{cases} 1, & \text{if } c1 = c2, \\ 0, & \text{otherwise.} \end{cases}$$

Example 3.2. (**Comparator**) *Figure 3.4 illustrates a possible bit-level implementation for the two-bits equivalence comparator with $o = f(a, b)$.*

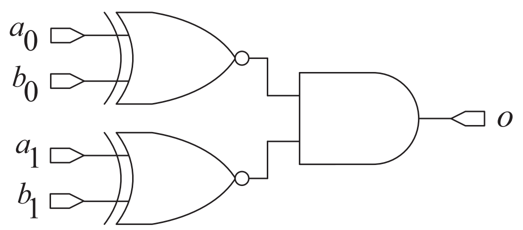


Figure 3.4: Two-bits comparator at gate level

3.1.4 ABL Description

Using the ABL objects introduced above, the ABL description can easily be formulated. However, before we do so, the meanings of the *fanin* and the *fanout* variables have to be explained.

Definition 3.5. (**Fanin/Fanout at ABL**) *Let F_1 be an ABL object with a set I_1 of input variables and a set O_1 of output variables. Here, $I_1 = \{i_{1_0}, \ldots, i_{1_n}\}$ and every i_{1_j} is called a fanin variable of F_1. For any $o_{1_k} \in O_1$, o_{1_k} is a fanout variable of F_1 iff there is another ABL object F_2 with I_2 and O_2, and it holds that $o_{1_k} \in I_2$.*

Example 3.3. (**Fanin/Fanout at ABL**) *Consider p_1 in Figure 3.1; this variable is a fanout of the partial product generator and a fanin of the addition network. Note that Definition 3.5 prohibits any fanout variables for comparators at ABL.*

Definition 3.6. (**ABL Description**) *Assume a set $\boldsymbol{P}$ of partial products, a set $\boldsymbol{N}$ of addition networks, and a set $\boldsymbol{C}$ of comparators. The ABL description is a DAG $G = (V, E)$ with the set of nodes $V = \boldsymbol{P} \cup \boldsymbol{N} \cup \boldsymbol{C}$ and the set of edges E such that for every $(x, y) \in E$ there exists a Boolean variable v which is a fanout for x and a fanin for y.*

It should be noted that every vertex of an ABL description stands for a Boolean function. Such a (potentially multi-output) function is defined for fanout variables and in terms of fanin variables of the vertex. Here, a function of each vertex is, therefore, restricted with the set of fanin variables. Thus, an ABL description consists of vertices with corresponding Boolean functions.

Due to the reason that topologies of ABL descriptions may differ for one and the same function, it is necessary to introduce a condition for the equivalence relation at the ABL.

Definition 3.7. *(***Equivalence Relation at ABL***) Two ABL descriptions G_1 and G_2 are equivalent with regard to a set S of Boolean variables iff for every $s_i \in S$, G_1 and G_2 represent the same Boolean function.*

Overall, an ABL description can be considered as a compact representation of arithmetic functions at the bit level. Apparently, the operations over ABL objects are straightforward and of low execution cost. Moreover, the representation of arithmetic functions at ABL is well suited in practice for a fast and reliable check on equivalence. This property of ABL descriptions makes them especially attractive for a verification in both equivalence and property checking. Therefore, the next section continues with a brief discussion on ABL-related approaches proposed in the past.

3.2 ABL in Formal Verification

In practice, the concepts of ABL serve as a good framework for a number of formal approaches to prove correctness of arithmetic designs in equivalence checking (Section 3.2.1) and in property checking (Section 3.2.2). Besides that, the concept of ABL was also successfully utilized for debugging of arithmetic circuits (Section 3.2.3).

3.2.1 Equivalence Checking at the ABL

ABL Extraction through a Reverse Engineering

The effectiveness of ABL reasoning with regard to formal verification was initially demonstrated in combinatorial equivalence checking to validate the absence of errors in designs of integer multipliers at the bit level, see [SK01] and [SK04] for thorough explanations on this topic.

The underlying idea of the proposed methods is based on the fact that, in a digital design flow, the gate-level representation is a synthesized description of the design after a series of optimizations performed at levels of abstraction lower than the word level or even lower than ABL. In other words, a functionally equivalent model at higher levels of abstraction needs to be derived from a given gate netlist. For purposes like this, the algorithms of [SK01] and [SK04] use *reverse engineering* guided by a half-adder network extraction. Here, Boolean reasoning techniques like, e.g., [KS97] are applied first to identify *XOR* functions inside the gate netlist. Further, these *XORs* are appended with appropriate carry signals detected from the logic elements of the gate-level design. Thus, the extraction procedure forms a reference circuit through an iterative insertion of the newly extracted half adders, see Figure 3.5, where the mapping of the output pins in HAs to the equivalent variables in the netlist is kept memorized. Further analysis of both the reference circuit and the gate netlist comes up with an extracted ABL model for the addition network.

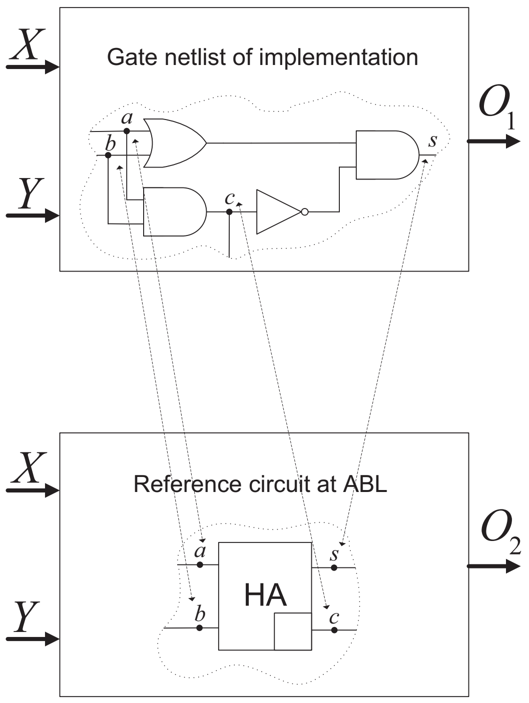


Figure 3.5: Mapping of a gate netlist to the reference circuit

However, the overall functionality of a multiplier design is not yet captured with such a reference circuit since it does not include a partial product generator as, e.g., shown in Figure 3.1. In practice, the choice of a suitable partial product generator depends on encodings used for the product bits in the implementation. The number of such encodings is not large and the matching of them to the given addition network is not a time-consuming task. The wrong encodings are usually canceled out in the first steps of the matching. By this fact, it allows to achieve a straightforward selection of the correct encoding without getting into any resource overheads for the whole extraction algorithm. Finally, a *merging operation*, as described in detail in Section 3.2.2, is applied to the reference circuit at the bit level. This leads to a straightforward proof that the reference circuit at ABL possesses all the desired functionality of a multiplier design. As a result, the reference circuit is well suited for the needs of a specification circuit used in a standard miter scheme of equivalence checking as depicted in Figure 1.1. Here, a specification circuit is applied to verify correctness of the gate-level implementation of a design. In other words, if it is proved that the functions defined at the output pins of the gate-level implementation are equivalent to the output functions of the specification then the design is correct. In our case, since the reference circuit contains a lot of internal equivalences with regard to the gate-level implementation, it is easier to perform the comparison of the output functions. In a case like this, the overall verification procedure is significantly faster.

As demonstrated in the experimental part of [SK01] and [SK04], the proposed extraction algorithm is quite robust in practice and it is able to tackle different multiplier designs optimized by commercial synthesis tools. Indeed, the algorithm is targeted at HA/FA searching inside a gate netlist. Unfortunately, in industrial applications, some optimizations may severely restructure the gate netlist so that, for some circuit regions, HA/FA networks cannot be extracted from the gate netlist. Example 3.4 describes this problem. An advanced approach to tackle problems like this is addressed in Chapter 5 where a more robust extraction algorithm is detailed.

Example 3.4. *(***Custom-Designed Component***) Assume a gate netlist of the full adder which is optimized because of the don't care conditions as defined in Table 3.1. In the context of the surrounding circuitry, the optimized FA behaves correctly because the inputs, for which the functions are don't care, are never applied to the FA. However, by a separate consideration of this piece of logic, also known as a custom-designed component, the extraction algorithm based on reverse engineering is infeasible to find any adequate ABL model for the optimized gate netlist of the full adder. This problem is illustrated in Figure 3.6.*

a	b	d	s	c
0	0	0	0	0
0	0	1	$*$	$*$
0	1	0	$*$	$*$
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	$*$	$*$
1	1	1	1	1

Table 3.1: Truth table for FA with some *don't care* conditions

In practice, optimizations caused due to don't care conditions may frequently occur in custom-designed components. Industrial RTL designs may contain a lot of diverse custom-designed entities. For example, a typical custom-designed component of an integer multiplier is a Booth encoder. This is an optimized at the logic level generator of all partial product bits. Figure A.6 illustrates the Verilog code of a multiplier design with such a generator.

Another technique to verify arithmetic circuits with the help of reverse engineering extraction was proposed in [SAF08]. This technique is in fact very similar to the procedure described above. However, in contrast to the algorithm of [SK01] and [SK04], where a stepwise extraction of HAs/FAs in a gate netlist is performed, the technique in [SAF08] aims to extract for each iteration a group of categorized HAs/FAs that forms a *bit-level adder (BLA)* in the reference circuit. The experimental part of [SAF08] compares the BLA technique against the conventional ABL algorithm. The authors of [SAF08] conclude with a superiority of the BLA approach. However, this result is experimentally not well supported because of the two reasons. First of all, for the experiments reported

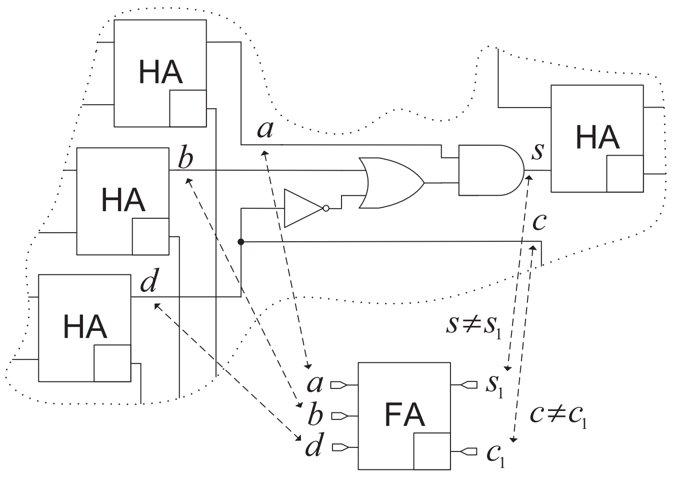


Figure 3.6: Custom-designed component problem at the ABL

in [SAF08], the authors used some private domain implementation of the ABL extraction. In contrast to that, in [SK01] and in [SK04], the ABL extraction algorithm was implemented as a part of the HANNIBAL tool [KS97]. Secondly, the mode of logic optimizations applied for the circuits tested in [SAF08] differs from the logic optimizations used for the benchmarks of [SK01, SK04]. The authors of [SAF08] optimized a gate netlist by means of Synopsys Design Compiler. However, in [SK01, SK04], the gate netlists generated by Synopsys Design Compiler were considered. It is evident that the degree of optimizations and restructuring is considerably higher for a gate netlist in the latter case. Therefore, the efficiency of the BLA technique with regard to the ABL-based algorithm still remains a point for discussion. Moreover, the problem of custom-designed components also seems to be a bottleneck for the approach of [SAF08].

Indeed, the extraction methodology described here is not the only way to obtain the ABL description from a gate-level representation of a design. As an alternative, the ABL model in terms of a reference circuit may also be created concurrently to the design and/or optimization process of a gate-level implementation. This approach is briefly described in the next section.

ABL Description through a Reference Language

The ABL extraction techniques reviewed in the previous section are based on reverse engineering. These techniques are applied to extract a reference circuit from a generated or optimized gate netlist of a design. Here, neither information about optimizations nor details of the design issues were considered. In reality, however, with knowledge about the design, such as encoding schemes for partial products or topologies used for addition networks, a reference circuit may not be extracted but, instead, simply generated. Such circuit will also expose a high amount of internal equivalences with respect to the gate-level implementation. This fact was the basic idea for the approach proposed in [KJW$^+$08]. Here, the authors introduce a special language, called *arithmetic reference* *description language (ARDL)*, aiming at two primary goals:

- an appropriate representation of the specification such that it can further serve as a reference circuit,
- regular updates of the specification to document all the necessary design issues applied for the gate-level implementation.

This language is syntactically very similar to the widely-used hardware description languages *VHDL* [Ash01] and *Verilog* [Bha99]. The language provides a small but sufficient set of combinatorial functions which allows a multiplier designer to detail the specification at an abstract level. This level is intended to be more detailed than the word level and a little more abstract than the gate level. Thus, the designer is able to start with the specification at earlier stages of the design process and always keep the specification up-to-date preserving as many internal equivalences as possible.

Finally, the specification is directly used to verify the correctness of the implementation as follows:

- the arithmetic correctness of the reference circuit is checked at the ABL with the algorithms as, e.g., described in Section 3.2.2, and in case of any errors they have to be fixed in the circuit,
- the gate netlist generated from the reference circuit is further used as a specification in a standard *miter scheme* of Figure 1.1 where the absence of errors in the gate-level representation has to be proved.

Due to the existence of internal equivalences between the specification and the implementation, the equivalence checking procedure performs very fast in practice on even large multiplier instances with input operands of large bit width such as 32, 64 bits.

However, the proposed ARDL-based technique is, in practice, oriented on full-custom applications. In case of a design with heavily optimized custom-designed components, the immediate translation of the RTL description into an ABL description for these components is impossible. Thus, a more powerful ABL extraction approach is needed such as the one discussed in Chapter 5.

3.2.2 Property Checking at the ABL

In this section ABL techniques for RTL property checking will be described. It should be noted that the methods of equivalence checking, reviewed in Section 3.2.1, extract or generate an ABL description from the gate netlist of a design. However, in standard RTL property checking, see Figure 1.2, whenever a design at the ABL is considered, the ABL description can easily be provided with the front-end of an RTL synthesis tool. An example of such a design is given in Figure A.5 as Verilog code for an integer multiplier.

In practice, an ABL description can be used to speed up property checking. For example, as shown in [WSK04, PWSK07], ABL reasoning may be used to prune the search space for SAT solving. Thus, a property checker with integrated DPLL-based algorithms can greatly be enhanced in solving arithmetic decision problems. In the following section we review another ABL-based technique, namely ABL normalization. This technique allows to prove correctness of arithmetic parts in a property checking instance so that the instance becomes significantly simplified.

ABL Normalization

As it was already mentioned above, SAT solving has de facto become a substantial ingredient of any modern RTL property checker. In practice, such tools perform well on control parts of a design and for detection of bugs. However, they usually fail to prove correctness of arithmetic functions in large industrial designs.

Problems like this are addressed by the *ABL normalization* technique proposed for the first time in [WSK05]. This technique proved to be very powerful; for example, all arithmetic operations in the data-path design of the industrial processor *TriCore* were successfully verified.

This section provides a brief summary on the ABL calculus for ABL normalization. In [WSBK07] the reader finds a more detailed description of this effective approach.

The underlying principle of the ABL normalization is based on a step-wise calculation of normal forms for arithmetic expressions. The normal forms are especially well suited to implement an efficient comparison of two arithmetic expressions. For normalized expressions, the comparison by subtraction of common terms can be applied. This is demonstrated in the following example.

Example 3.5. *(***Normalization***) Assume an arithmetic logic unit with an addition / multiplication operation on two input operands $a = (a_0, a_1, a_2)$ and $b = (b_0, b_1, b_2)$. The result of the operation can be described with the following expression $r = a(a + b) = (a_0+2a_1+4a_2)((a_0+2a_1+4a_2)+(b_0+2b_1+4b_2))$. Suppose that the reduced normalized form of the expression r is assumed to be as follows $r'_p = a_0 + a_0b_0 + 2a_0b_1 + 2a_1b_0 + 4a_1 + 4a_0a_1 + 4a_0b_2 + 4a_1b_1 + 4a_2b_0 + 8a_0a_2 + 8a_1b_2 + 8a_2b_1 + 16a_2 + 16a_1a_2 + 16a_2b_2$. To enable the test for functional equivalence between r and r'_p, it is sufficient to calculate the normal form r_p for r and check whether the equation $r_p = r'_p$ holds. Since both expressions r_p and r'_p consist of a sum of terms, the comparison by subtraction of common terms becomes trivial for the equation $r_p - r'_p = 0$.*

In contrast to Example 3.5, unfortunately, a trivial normalization approach is not applicable for a real design, since the arithmetic functions of the design are implemented by a network of half adders and/or full adders. Before we start to describe in details the more sophisticated technique of [WSK05], some additional definitions and explanations are needed. We will provide them by referring to the notations given in Section 3.1 and in [WSBK07]. First of all, the normal form for an ABL description has to be defined.

Definition 3.8. *(***ABL Normal Form***) An ABL description $(\{\boldsymbol{P} \cup \boldsymbol{N} \cup \boldsymbol{C}\}, E)$ is expressed in a normal form iff there is no edge $(N, P) \in E$ for any $N \in \boldsymbol{N}$ and for any $P \in \boldsymbol{P}$.*

Definition 3.9. *(***Reduced ABL Normal Form***) A normal form ABL description $(\{\boldsymbol{P} \cup \boldsymbol{N} \cup \boldsymbol{C}\}, E)$ is called reduced iff it always holds that:*

- *two addition networks $N_1 \in \boldsymbol{N}$ and $N_2 \in \boldsymbol{N}$ share no common addends if there exist $C \in \boldsymbol{C}$ such that $(N_1, C) \in E$ and $(N_2, C) \in E$,*
- *there is no edge $(N_1, N_2) \in E$ for any $N_1, N_2 \in \boldsymbol{N}$.*

For a more compact representation, the normal ABL form should also be reduced so that it conforms to Definition 3.9. However, note, such a normal form is not unique.

A normal form calculation would be impossible at the ABL without proper operations over ABL elements. Intuitively, such operations have to obey arithmetic commutative, associative and distributive laws. Definitions 3.10 - 3.12 introduce three major operations that constitute a fundamental basis for the ABL normalization technique.

Definition 3.10. *(***Merging of Addition Networks***) The addition network N_3 is a result of merging an addition network N_1 with some addition network N_2 if and only if the ABL description $(V, E) = (\{N_3\}, \emptyset)$ is equivalent to the ABL description $(V', E') = (\{N_1, N_2\}, \{(N_1, N_2)\})$.*

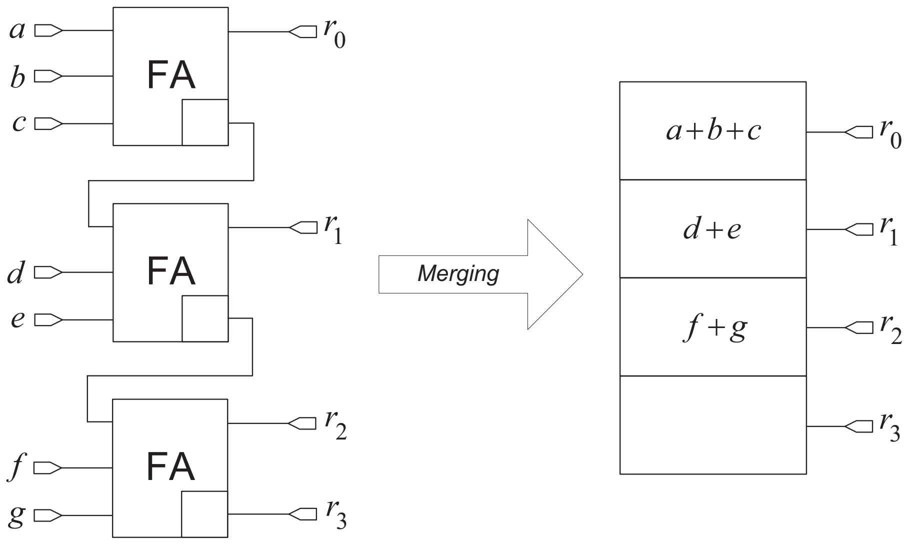


Figure 3.7: Merging of addition networks at ABL

Example 3.6. *(***Merging***) Figure 3.7 demonstrates an example of merging at ABL, where three FAs of the carry-ripple adder, shown on the left-hand side, are merged into the single four-column addition network on the right-hand side.*

Besides the compactness of multi-column addition networks at ABL, Example 3.6 also illustrates another beneficial feature of such an ABL model, namely the carry bits are not explicitly defined, but, they are always implicitly assumed. Moreover, merging of two addition networks can be done in linear time with regard to a number of addends.

Similar to the merging of addition networks, one may also define a merging of partial products, see Definition 3.11, which is of linear complexity as well.

Definition 3.11. *(***Merging of Partial Products***) Two partial product generators P_1, P_2 are merged into a partial product generator P_3 if and only if it holds that for any $p_{1_i} \in P_1$ there exists $p_{3_n} \in P_3$ with $p_{3_n} = p_{1_i}$ and for any $p_{2_j} \in P_2$ there exists $p_{3_m} \in P_3$ with $p_{3_m} = p_{2_j}$.*

According to Definition 3.8, a normal form at ABL is not allowed to contain any addition networks whose result bits are in fanin to partial products. In other words, all partial products that are in the fanout to an addition network have to be moved in front of this network so that the functionality of the overall ABL description is preserved. For purposes like this, the operation of partial products distribution is applied, see Definition 3.12.

Definition 3.12. *(***Partial Products Distribution***) Let (V, E) be an ABL description with an addition network N and a partial product generator P such that $(N, P) \in E$. It is possible to distribute P through N iff there exist a partial product generator P' and addition networks $N'_1, \ldots, N'_m$ that add up all the partial products of P' so that the substitution of (N, P) with P' and $N'_1, \ldots, N'_m$ results in the equivalent ABL description.*

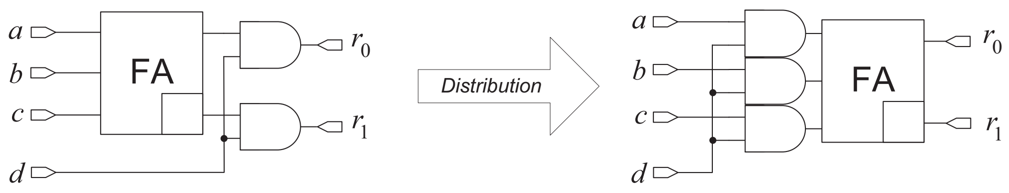
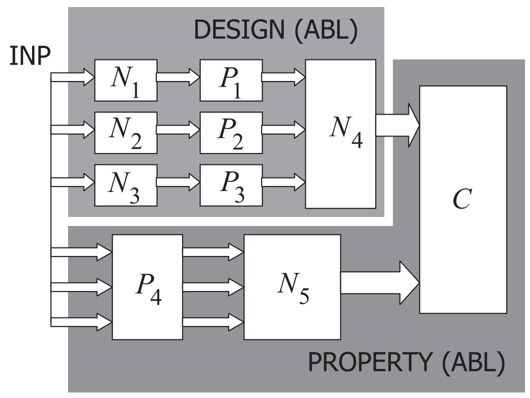


Figure 3.8: Distribution at ABL

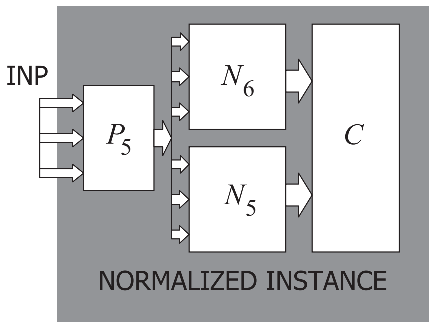
An example of a partial products distribution through a full adder is depicted in Figure 3.8. In fact, a distribution at ABL generates an m-times replication of the addition network, where m is the size of that operand in P which is multiplied out with the result operand of N, see Definition 3.12. Such replications may become computationally costly. In practice, however, the newly generated networks are usually immediately merged, avoiding CPU time blow-up.

The flow of the ABL normalization is schematically depicted in Figure 3.9. According to the definitions of this section, it can be formulated as follows:

- merging of addition networks,
- merging of partial products,
- distribution of partial products,



(a) ABL instance formed from design and property



(b) ABL normalized instance

Figure 3.9: ABL normalization flow

- merging of addition networks,
- identification of equivalent partial products,
- deletion of equivalent addends from the addition networks that are in the fanin to the common comparator.

For a thorough example on the ABL normalization, as well as experimental results, the interested reader is referred to [WSBK07].

The theory of the ABL calculus, briefly summarized in this section, serves as a background for the ABL verification method based on the computer algebra technique presented in Chapter 4.

3.2.3 Debugging by means of ABL Description

Along with a successful application in the domain of formal verification, the principles of the ABL extraction [SK01] turned out to be useful in the debugging of gate netlists for arithmetic designs.

For example, the approach [KF02] exploits an ABL reference model to identify bugs in the erroneous part of a design. Here, the circuit in question is mapped to standard logic cells. In case of any errors detected in the addition network of the circuit, the erroneous part is extracted for the further consideration. This part is transformed into a net of primitive logic gates so that the mapping to a reference model, as described in [SK01], can be applied. The process of this mapping is divided into two phases, namely a forward mapping and a backward mapping, i.e., when the circuit is traversed from the primary inputs towards the primary outputs first and, then, in the opposite direction. Such a mapping locates bugs by the unmapped piece inside the erroneous circuitry. As soon as the faulty elements are identified, they are replaced in a proper manner.

Recently, Sarbishei et al. proposed another formal approach for the debugging of gate-netlist descriptions of arithmetic circuits [STAF09]. Again, the key feature of the approach is based on a reverse engineering that treats a gate netlist as a target to extract an ABL model for the needs of the debugging. The algorithm for the debugging itself consists of three major steps:

- initialization of partial products,
- extraction of XOR functions,
- mapping for carry signals.

Like in [KF02], the approach of [STAF09] provides a mechanism for rectification of the erroneous parts in the circuit of interest.

Chapter 4

Algebraic Approach

In this chapter we study a recently developed technique [WWS$^+$08, PWS$^+$11] for solving hard arithmetic problems. This technique is based on the theory of Gröbner bases over finite rings. First, we convert the arithmetic parts of a decision problem into equivalent *variety subset problems*. Then, we solve these problems by computing normal forms. Contrary to [WWS$^+$08], we now perform our computations in the quotient ring $Q := \mathbb{Z}/2^N[X]/\langle x^2 - x \ : \ x \in X\rangle$. We prove that this allows us to omit an otherwise necessary and expensive zero function test for the normal form.

This chapter consists of three sections. Section 4.1 provides a short review on previous algebraic techniques used in formal verification of data-path designs. Section 4.2 shows how the arithmetic subproblems of the instance under interest can efficiently be solved using computer algebra techniques. In this section, we assume that arithmetic problem parts are described at the arithmetic bit level or at the word level. However, for cases where problem parts are specified below ABL, the extraction techniques discussed in Chapter 5 need to be applied first. In Chapter 6 we combine these approaches and provide experimental results. This chapter ends with a representative example discussed in Section 4.3. Here, using the proposed approach based on computer algebra, the proof of correctness for a multiplier design is illustrated.

4.1 Related Work

Recently, techniques from symbolic computer algebra have entered the verification arena [SKEG05, SKE06, SKE07, WHAH07, SKME08, WWS$^+$08, BDG$^+$09, PWS$^+$11]. For instance, in [SKEG05], a procedure is presented that determines whether a multivariate polynomial with fixed word length operands is vanishing. This allows to compare polynomial representations for bit vector functions. The approach is extended towards multiple word length operands in [SKE06, SKE07]. Both approaches, however, require a clean word-level representation of the data paths. This limits their applicability, e.g., in RTL property checking.

In principle, it is possible to transform a circuit-related verification problem into an algebraic problem over $\mathbb{Z}/2$. Efficient computer algebra systems for this special case are available [BD09]. In [BD09], polynomials are represented by *zero suppressed binary decision diagrams (ZDDs)* [Min93]. As a result, during the algebraic computation, *functional decision diagrams (FDDs)* [DBS$^+$94] of the original circuitry are generated. FDDs, however, are known to grow exponentially for multipliers and are, therefore, unsuitable for the problems considered in this thesis. At the bit level, arithmetic circuitry is typically specified using arithmetic entities such as half adders and full adders. If such entities can be identified within the design, we call the resulting netlist an arithmetic bit level (ABL) description. An approach for verification of such bit level implementations using Gröbner basis theory is reported in [WHAH07]. This approach requires polynomial specifications for every building block in the hierarchy of the arithmetic circuit design. After proving that a block, e.g., a carry-save adder, fulfills its local specification, the polynomial representation is used to verify the block in the next level of the hierarchy. However, since the correctness proof includes a range check, the intermediate results at the block boundary are required to have sufficient bit width to represent every possible result. For designs implementing integer arithmetic with fixed bit width, this is often not the case.

A heuristic approach to exploit arithmetic bit level information in RTL designs has been reported in [WSBK07]. By local equivalence transformation of the arithmetic bit level description, a reduced normal form is computed that is sufficient to prove the arithmetic problem parts of a property checking instance and relieves the SAT/SMT solver from reasoning in structurally different implementations for the same arithmetic function. This method is subsumed by more general algebraic approaches, e.g., as presented in [WWS$^+$08, BDG$^+$09, PWS$^+$11], and, further, investigated in the next section of this chapter. Besides, these approaches also provide a well-understood mathematical basis for ABL-based verification techniques reviewed in Section 3.2.2.

4.2 Using Computer Algebra to solve Arithmetic Subproblems in Formal Verification

This section starts with an overview about mathematical models required to formulate an arithmetic decision problem for the algebraic technique introduced in [WWS$^+$08]. The normal form computations for solving such problems are discussed in the sequel.

4.2.1 Algebraic Modeling of Arithmetic Decision Problems

We may assume that the arithmetic decision problem is represented as an acyclic netlist of bit vector functions. The arithmetic components G_j of this netlist may have multiple output bit-widths that are denoted by n_j in the sequel. In [WWS$^+$08], each of these components is modeled by n_j polynomials over the ring $R = \mathbb{Z}/2^N$ with appropriate N. Initially, the size N of the ring is chosen heuristically to be

$$N := n + \max\{n_j \,|\, j = 1, \ldots, m\} \,,$$

where n is the bit-width of the comparison constraint in the proof goal, m the component number in the netlist and n_j the bit-width of the j-th component. However, this initial value for N is just a heuristic choice that turned out to be sufficient for many practical problem instances. During computations, our current implementation detects cases where a larger ring is required and automatically moves to a sufficiently larger ring with size $N' > N$.

We can describe each arithmetic component in the cone of influence of a proof goal in terms of n_j equations of the form

$$G_j^{(t)} : \sum_{i=0}^{t-1} 2^i r_i^{(j)} = f_j^{(t)}(a_1^{(j)}, a_2^{(j)}, \ldots, a_{m_j}^{(j)}) \bmod 2^t, \tag{4.1}$$

with $t = 1, \ldots, n_j$. Moreover, the variables $a_i^{(j)}$ correspond to the inputs and the variables $r_i^{(j)}$ correspond to the outputs of the j-th component. The polynomials $f_j^{(t)}$ are defined as the polynomials with minimal coefficients representing the polynomial function $(\mathbb{Z}/2^t)^{m_j} \to \mathbb{Z}/2^t$ that specifies the lower most output bits $r_i^{(j)}$ for $i = (0, \ldots, t-1)$ of the arithmetic component G_j. Furthermore, the condition $r_i^{(j)} \neq a_k^{(l)}$ is fulfilled by definition, since the netlist is acyclic.

As the reader might have noticed, the modulo operation used in Equations 4.1 is not an algebraic operation. In order to apply Equations 4.1 for algorithms of computer algebra, we have to model the modulo semantics of the arithmetic components in bit-vector netlists. For this reason, artificial variables $s_t^{(j)}$ – so-called *slack variables* – are additionally introduced. Thus, from Equations 4.1 we generate n_j polynomials as follows:

$$\tilde{G}_j^{(t)} : \sum_{i=0}^{t-1} 2^i r_i^{(j)} - f_j^{(t)}(a_1^{(j)}, a_2^{(j)}, \ldots, a_{m_j}^{(j)}) - 2^t s_t^{(j)}. \tag{4.2}$$

These polynomials are sufficient to model word-level arithmetic components like signed-/unsigned multipliers or adders. Likewise, bit-level arithmetic components such as full adders, half adders, and bit-wise products can be modeled in the same formalism. We illustrate these definitions by the following examples.

Example 4.1. *(**Partial Products**) Given a non-Booth-encoded $(n \times m)$ multiplier with two bit vectors $(a_0, \ldots, a_{n-1})$ and $(b_0, \ldots, b_{m-1})$ as input operands. The partial products of such a multiplier can be represented in terms of k equations as defined in Formula 4.3 and, then, be modeled by k polynomials of Formula 4.4, where $k = n \cdot m$; $t = 0, \ldots, k-1$; $i = 0, \ldots, n-1$ and $j = 0, \ldots, m-1$.*

$$G_{pp}^{(t)} : p_{i,j} = a_i b_j \bmod 2. \tag{4.3}$$

$$\tilde{G}_{pp}^{(t)} : p_{i,j} - a_i b_j + 2s_1. \tag{4.4}$$

Example 4.2. (**Half Adder**) *A half adder with inputs x_0, x_1 and outputs c, s for carry and sum can be represented by Equations 4.5 and, then, be modeled by the polynomials of Formula 4.6.*

$$\begin{aligned} G_{ha}^{(2)} &: 2c + s = (x_0 + x_1) \bmod 4, \\ G_{ha}^{(1)} &: s = (x_0 + x_1) \bmod 2. \end{aligned} \tag{4.5}$$

$$\begin{aligned} \tilde{G}_{ha}^{(2)} &: 2c + s - (x_0 + x_1) + 4s_2, \\ \tilde{G}_{ha}^{(1)} &: s - (x_0 + x_1) + 2s_1. \end{aligned} \tag{4.6}$$

Example 4.3. (**Full Adder**) *A full adder with inputs x_0, x_1, x_2 and outputs c, s for carry and sum can be represented by Equations 4.7 and, then, be modeled by the polynomials of Formula 4.8.*

$$\begin{aligned} G_{fa}^{(2)} &: 2c + s = (x_0 + x_1 + x_2) \bmod 4, \\ G_{fa}^{(1)} &: s = (x_0 + x_1 + x_2) \bmod 2. \end{aligned} \tag{4.7}$$

$$\begin{aligned} \tilde{G}_{fa}^{(2)} &: 2c + s - (x_0 + x_1 + x_2) + 4s_2, \\ \tilde{G}_{fa}^{(1)} &: s - (x_0 + x_1 + x_2) + 2s_1. \end{aligned} \tag{4.8}$$

Example 4.4. (**Adder**) *An unsigned k-bit adder with input variables $a = (a_i \mid 0 \leq i < k)$ and $b = (b_i \mid 0 \leq i < k)$, and result $r = (r_i \mid 0 \leq i < k)$ can be represented by k equations $G_+^{(t)}$ stated with Formula 4.9, where $t = 1, \ldots, k$, and, further, be modeled by k polynomials $\tilde{G}_+^{(t)}$ of Formula 4.10.*

$$G_+^{(t)} : \sum_{i=0}^{t-1} 2^i r_i = \sum_{i=0}^{t-1} 2^i (a_i + b_i) \bmod 2^t. \tag{4.9}$$

$$\tilde{G}_+^{(t)} : \sum_{i=0}^{t-1} 2^i r_i - \sum_{i=0}^{t-1} 2^i (a_i + b_i) - 2^t s_t. \tag{4.10}$$

Example 4.5. (**Multiplier**) *An unsigned $(n \times m)$-bit multiplier with input variables $a = (a_x \mid 0 \leq x < n)$ and $b = (b_y \mid 0 \leq y < m)$, and result $r = (r_i \mid 0 \leq i < k)$, where $k = n + m$, can be represented by k equations $G_\times^{(t)}$ stated with Formula 4.11, where $t = 1, \ldots, k$, and, further, be modeled by k polynomials $\tilde{G}_\times^{(t)}$ of Formula 4.12.*

$$G_\times^{(t)} : \sum_{i=0}^{t-1} 2^i r_i = \sum_{x=0}^{n-1} 2^x a_x \cdot \sum_{y=0}^{m-1} 2^y a_y \bmod 2^t. \tag{4.11}$$

$$\tilde{G}_\times^{(t)} : \sum_{i=0}^{t-1} 2^i r_i - \sum_{x=0}^{n-1} 2^x a_x \cdot \sum_{y=0}^{m-1} 2^y a_y - 2^t s_t. \tag{4.12}$$

In the polynomials $\tilde{G}_j^{(t)}$, some of the slack variables $s_t^{(j)}$ can be omitted if we know that the condition defined with Formula 4.13 holds over $\mathbb{Z}$.

$$0 \leq f_j^{(t)} \leq 2^t - 1. \tag{4.13}$$

For example, the slack variable s_1 of Formula 4.4 and the slack variables s_2 in Formula 4.12 and in Formula 4.6 can be omitted. If the condition of Formula 4.13 cannot be guaranteed and we need to know the exact value of $s_t^{(j)}$ during the computation, we can replace $s_t^{(j)}$ by a polynomial in the variables $a_1^{(j)}, a_2^{(j)}, \ldots, a_{m_j}^{(j)}$, i.e., a subset of the inputs of G_j. For instance, the polynomial $s - (x_0 + x_1) + 2s_1$ of Formula 4.6 results in the polynomial $s_1 = x_0 \cdot x_1$ for the slack variable. However, often, it is better to introduce slack variables because, in general, the polynomials for such variables can be very large even for small polynomials for $f_j^{(t)}$.

As illustrated in the examples, the above algebraic model utilizes word-level information where available and is able to handle bit-level arithmetic information where necessary as well.

In order to analyze proof goals with algebraic methods, the notion of variety as given in Definition 2.24 is needed.

Let X be the set of non slack variables in $\{\tilde{G}_j^{(t)}\}$. A proof goal is modeled by a polynomial g in $\{a_1, \ldots, a_t\} \subset X$. It vanishes if and only if the proof goal is fulfilled, i.e., the following condition

$$g(a_1, \ldots, a_t) = 0 \bmod 2^n$$

holds for all tuples in the variety $V(\{\tilde{G}_j\})$. Note that n, used for the modulo in the above equation, depends on the bit width of the comparison constraint in the underlying problem instance.

Example 4.6. *(***Equality Comparison***) A n-bit equality comparison of operands $a = (a_{n-1}, \ldots, a_0)$ and $b = (b_{n-1}, \ldots, b_0)$ can be modeled by the polynomial of Equation 4.14.*

$$g = \sum_{i=0}^{n-1} 2^i (a_i - b_i)\,. \tag{4.14}$$

To decide whether the condition $g = 0$ modulo 2^n holds, we have to prove that the polynomial of Equation 4.14 vanishes modulo 2^n for all tuples in the variety $V(\{\tilde{G}_j\})$. This leads to the *variety subset problem*:

$$V(\{\tilde{G}_j\}) \subset V(2^{N-n} g)\,. \tag{4.15}$$

Following, we solve this for $R = \mathbb{Z}/2^N$. Our solution replaces the well-known *ideal membership problem* of the field case.

Note that the generated variety subset problem can in principle be solved by constructing a Gröbner basis for the ideal $I = \langle\{\tilde{G}_j\}\rangle$ and using normal form computations with respect to this basis. Fortunately, it turns out that for certain monomial orderings, the set $\{\tilde{G}_j\}$ is a Gröbner basis for I by construction. This is crucial for the performance of the presented approach.

4.2.2 Solving Arithmetic Decision Problems by Normal Form Computations

In this section, Proposition 4.1 and Lemma 4.1, defined in [WWS$^+$08] at first, are key aspects in an effective solution of the variety subset problem introduced in the previous section.

Proposition 4.1. *The set $G = \{\tilde{G}_j^{(t)}\}$ is a Gröbner basis with respect to any global monomial ordering refining the following partial order:*

$$r_i^{(j)} \text{ is larger than every monomial in the variables } a_k^{(j)}, s_t^{(j)}, r_l^{(j)},$$

for all i, k, t, j and $l < i$.
Proof: *see [WWS$^+$08].* □

Indeed, as shown in [WWS$^+$08], Lemma 4.1 serves as a basis to prove that normal form computation can be used as an effective procedure for solving our variety subset problems.

Lemma 4.1. *Let G be a Gröbner basis of an ideal $I \subset \mathbb{Z}/2^N[X], X = (X', X'')$, and assume that for all X' there exist X'' with $f(X', X'') = 0$ for all $f \in G$. Let g be a polynomial such that the normal form h of g with respect to G is in $\mathbb{Z}/2^N[X']$. Then, h defines the zero function if and only if $V(G) \subset V(g)$.*
Proof: *see [WWS$^+$08].* □

For the given proof goal $g \in \mathbb{Z}/2^n[X]$, the normal form $h =$NF$(2^{N-n}g \mid G)$ of $2^{N-n}g$ with respect to G can efficiently be computed, e.g., by the algorithm depicted in Figure 2.7. It holds if and only if h defines the zero function, since we may assume that h depends on inputs only.

Returning to the final statement of Lemma 4.1, one should note that although h defines the zero function, it need not be the zero polynomial; cf. [GSW10]. Moreover, h may initially consist of slack variables, and replacing slack variables by polynomial expressions in the inputs may significantly increase the complexity, as pointed out above.

Thus, the goal is to move to a modified setting in which Lemma 4.1 will still hold but slack variables and non-zero polynomials defining the zero function are no longer a problem. In general, zero function tests over $\mathbb{Z}/2^N[X]$ are expensive [SKE07].

A positive result in this respect is obtained by using a pure bit-level formulation, that is, all variables in Equation 4.1 and in Equation 4.2 are modeled to be bit-valued. This can be enforced by adding extra relations of the form $x^2 = x$, for all variables in X. Algebraically, this means to compute in the quotient ring $Q := \mathbb{Z}/2^N[X]/\langle x^2 - x \ : \ x \in X\rangle$ rather than in $\mathbb{Z}/2^N[X]$. Lemma 4.2 ensures that only the zero polynomial in Q defines the zero function on $Q^{|X|}$ and renders the zero function test superfluous here. Hence, the precondition of Lemma 4.1 for concluding $V(G) \subset V(g)$ can be effectively tested.

Lemma 4.2. *Let $m, n \geq 1$ be natural numbers and $Q := \mathbb{Z}/\langle m\rangle[x_1, x_2, \ldots, x_n]/\langle x_1^2 - x_1, x_2^2 - x_2, \ldots, x_n^2 - x_n\rangle$ be a polynomial quotient ring. Moreover, let us denote by $T := \{(t_1, t_2, \ldots, t_n) \mid t_i \in \{0, 1\}, 1 \leq i \leq n\}$ the set of all bit-valued inputs for polynomials in Q. If $f \in Q$ vanishes for all $t \in T$, then f is the zero polynomial.*

Proof: *We fix some $f \in Q$ which vanishes for all $t \in T$, and assume $f \not\equiv 0$ for a contradiction. Due to special structure of Q, all terms of f are of the form $c \cdot x_{i_1} \cdots x_{i_k}$ with mutually distinct indices i_j in $\{1, 2, \ldots, n\}$. We pick a term s of f with k least, i.e., with the least number of variables. Now, consider the tuple $t = (t_1, t_2, \ldots, t_n) \in T$ with $t_j = 1$ if x_j appears in s, and $t_j = 0$ otherwise. We claim $f(t) \neq 0$, yielding the desired contradiction. Let s' denote any other term of f, i.e., any term of $f - s$. By construction, it is clear that s' mentions at least one variable which is not present in s. But then $s'(t) = 0$, hence, $f(t) = s(t) + (f - s)(t) = s(t) = LC(s) \neq 0$.* □

4.3 Illustrative Example

In this section we demonstrate an application of the computer algebra method described above in this chapter. As an example, we show how the design of a (2×2) signed integer multiplier depicted in Figure 4.1 can be verified. We use the generic computer algebra tool *SINGULAR 3-1-2* [GPS10] to perform the computations.

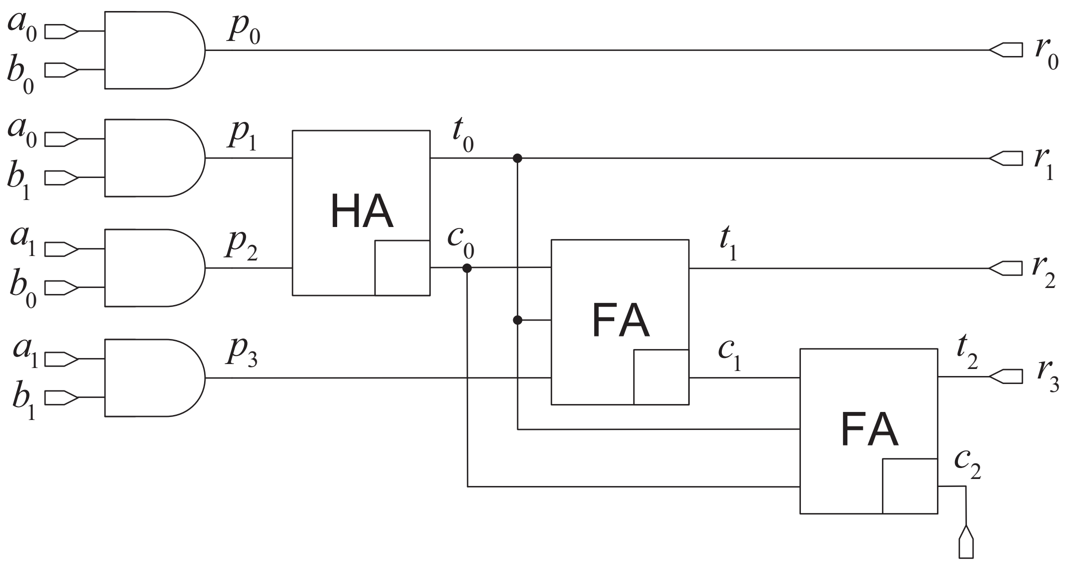


Figure 4.1: (2×2) signed integer multiplier

#	unit	functional definition	algebraic polynomial(s)
1	*AND*	$p_0 = a_0 \wedge b_0$	$p_0 - a_0b_0$
2	*AND*	$p_1 = a_0 \wedge b_1$	$p_1 - a_0b_1$
3	*AND*	$p_2 = a_1 \wedge b_0$	$p_2 - a_1b_0$
4	*AND*	$p_3 = a_1 \wedge b_1$	$p_2 - a_1b_1$
5	*HA*	$t_0 + 2c_0 = (p_1 + p_2) \bmod 2^2$	$t_0 - (p_1 + p_2) + 2s_0,$ $t_0 + 2c_0 - (p_1 + p_2)$
6	*FA*	$t_1 + 2c_1 = (t_0 + c_0 + p_3) \bmod 2^2$	$t_1 - (t_0 + c_0 + p_3) + 2s_1,$ $t_1 + 2c_1 - (t_0 + c_0 + p_3)$
7	*FA*	$t_2 + 2c_2 = (t_0 + c_0 + c_1) \bmod 2^2$	$t_2 - (t_0 + c_0 + c_1) + 2s_2,$ $t_2 + 2c_2 - (t_0 + c_0 + c_1)$
8	*OUTPUT*	$r_0 = p_0$	$r_0 - p_0$
9	*OUTPUT*	$r_1 = t_0$	$r_1 - t_0$
10	*OUTPUT*	$r_2 = t_1$	$r_2 - t_1$
11	*OUTPUT*	$r_3 = t_2$	$r_3 - t_2$

Table 4.1: Mathematical description of units for multiplier from Figure 4.1 where all variables are Boolean variables and, moreover, s_0, s_1, s_2 define slack variables

functional definition	algebraic polynomials
$r'_0 + 2r'_1 + 4r'_2 + 8r'_3 =$ $(-2a_1 + a_0)(-2b_1 + b0) =$ $a_0b_0 - 2a_0b_1 - 2a_1b_0 + 4a_1b_1$	$r'_0 - a_0b_0,$ $r'_0 + 2r'_1 - (a_0b_0 - 2a_0b_1 - 2a_1b_0) + 4s_3,$ $r'_0 + 2r'_1 + 4r'_2 - (a_0b_0 - 2a_0b_1 - 2a_1b_0 + 4a_1b_1) + 8s_4,$ $r'_0 + 2r'_1 + 4r'_2 + 8r'_3 - (a_0b_0 - 2a_0b_1 - 2a_1b_0 + 4a_1b_1)$

Table 4.2: Mathematical description of a (2×2) signed multiplier where all variables are Boolean variables and, moreover, s_3, s_4 define slack variables

#	commands	comments
1	ring $r = (\text{integer}, 2, 4), (r_3, r_2, r_1, r_0, r'_3, r'_2, r'_1,$ $r'_0, c_2, t_2, c_1, t_1, c_0, t_0, p_3, p_2, p_1, p_0, a_1, a_0, b_1, b_0,$ $s_4, s_3, s_2, s_1, s_0), lp;$	Define the ring r of integers modulo 2^4 with lexicographical ordering. Note, the variables are topologically sorted, see Figure 4.1.
2	ideal $I =$ $p_0 - a_0 * b_0,$ $p_1 - a_0 * b_1,$ $p_2 - a_1 * b_0,$ $p_3 - a_1 * b_1,$ $t_0 - p_1 - p_2 + 2 * s_0,$ $t_0 + 2 * c_0 - p_1 - p_2,$ $t_1 - c_0 - t_0 - p_3 + 2 * s_1,$ $t_1 + 2 * c_1 - c_0 - t_0 - p_3,$ $t_2 - t_0 - c_0 - c_1 + 2 * s_2,$ $t_2 + 2 * c_2 - t_0 - c_0 - c_1,$ $r_0 - p_0,$ $r_1 - t_0,$ $r_2 - t_1,$ $r_3 - t_2,$ $r'_0 - a_0 * b_0,$ $r'_0 + 2 * r'_1 - (a_0 * b_0 -$ $2 * a_1 * b_0 - 2 * a_0 * b_1) + 4 * s_2,$ $r'_0 + 2 * r'_1 + 4 * r'_2 - (a_0 * b_0 - 2 * a_1 * b_0$ $-2 * a_0 * b_1 + 4 * a_1 * b_1) + 8 * s_3,$ $r'_0 + 2 * r'_1 + 4 * r'_2 + 8 * r'_3 - (a_0 * b_0$ $-2 * a_1 * b_0 - 2 * a_0 * b_1 + 4 * a_1 * b_1);$	Generate ideal I from algebraic polynomials of Table 4.1 and Table 4.2.
3	reduce(lead(std(I)),std(lead(I)));	Check that I is a Gröbner basis. This is true if the reduction results in zero for all polynomials of I.
4	ideal J; for (int $i = 1; i <=$nvars$(r); i++)$ $\{J = J,$ideal(var$(i)^2-$var$(i)); \};$	Generate ideal J to ensure that the condition $x^2 = x$ holds for any variable x of the ring r, i.e., x is a Boolean variable.
5	reduce(lead(std(J)),std(lead(J)));	Check that J is a Gröbner basis.
6	poly $g =$ $r_0 + 2 * r_1 + 4 * r_2 + 8 * r_3$ $-(r'_0 + 2 * r'_1 + 4 * r'_2 + 8 * r'_3);$	Define the proof goal.
7	reduce(reduce(g, I), J);	Compute reduced normal form of g with respect to I and J. This computation results in zero.

Table 4.3: Normal form computation with SINGULAR 3-1-2 [GPS10]

As pointed out in Section 4.2, to apply the computer algebra technique, first of all, it is necessary to derive algebraic polynomials for each unit of the design under investigation. For the net list of the (2×2) signed multiplier from Figure 4.1, the mathematical definitions and the polynomials for every unit are shown in the second and the third column, respectively, of Table 4.1. Furthermore, Table 4.2 contains the definition of all corresponding polynomials for the (2×2) signed multiplier. Note that in the multiplier of Figure 4.1 the bit vectors (a_1, a_0) and (b_1, b_0) are used as primary inputs whereas the bit vector (r_3, r_2, r_1, r_0) serves as primary output. The vector (r'_3, r'_2, r'_1, r'_0) stands for the output function of a word-level (2×2) signed multiplier given in the specification. A few slack variables s_3, s_2, s_1, s_0 are needed. The conversion of logic functions into algebraic polynomials is implemented according to the technique described in Section 5.3.1. The transformation of ABL elements into a set of algebraic polynomials follows to the modeling approach of Section 4.2.1.

Thus, the proof goal of our example can algebraically be formulated as the polynomial $g = r_0 + 2r_1 + 4r_2 + 8r_3 - (r'_0 + 2r'_1 + 4r'_2 + 8r'_3)$. If we prove that $g = 0$ for all possible evaluations of the primary input variables then the design of the multiplier in Figure 4.1 is correct. To enable computations over algebraic polynomials, SINGULAR 3-1-2 [GPS10] is used. All relevant steps of the computations are gathered in Table 4.3. At first, the ring r of integers modulo 2^4 is defined. Here, all variables of the created algebraic polynomials are specified with regard to topological ordering. In the next step, the ideal I is generated over all the created polynomials. In accordance with the theory of [WWS$^+$08], the ideal I has to form a Gröbner basis. To check this, it is enough to prove whether the condition of Definition 2.27 holds. Therefore, a special command of the third step, shown in Table 4.3, is applied for SINGULAR. For our example, this command produces zero for all the reduced polynomials of the ideal I. Since all the variables used for polynomials are Boolean variables, the condition $x^2 = x$ has to be always fulfilled for them, see Lemma 4.2. Due to this, an additional ideal J is generated at the step 4. Actually, the ideal J is also a Gröbner basis. This is shown in step 5. At the step 6, the proof goal is specified. Finally, at the step 7, the reduced normal form for the proof goal is calculated with respect to Gröbner bases I and J. Based on the fact that the computed normal form is a constant zero, we conclude with the functional correctness for the design of the multiplier in Figure 4.1.

The computer algebra technique proposed in this section was integrated into the verification engine described in Chapter 6. We also present experimental data evaluating the efficiency of the technique.

Chapter 5

Modeling of Custom-Designed Components at the ABL

In this chapter, we increase the level of automation for both RTL property checking and SMT solving of QF-BV formulas in those cases where, for arithmetic circuits, the ABL information is missing for certain custom-made components of a design. Such components might be, e.g., Booth encoders or sophisticated addition structures that are typically implemented below the ABL. This is relevant for a large number of industrial applications where certain arithmetic components are custom-designed and others (especially array structures such as addition networks) are not. In practice, often, some parts of a design contain hand-crafted optimizations and include specialized logic such that it becomes impossible to translate the RTL description into an ABL description immediately. In this chapter, we study a method to transform custom-designed components of an arithmetic circuit into functionally equivalent ABL descriptions. Consequently, this allows to successfully apply the ABL-based algorithms described in Chapters 3, 4. The proposed approach fills an important gap in the formal verification flow and ensures that the manual approach [KJW$^+$08], described in Section 3.2.1, can be reserved for high-end applications where arithmetic circuits are designed globally by a full-custom methodology.

This chapter is organized as follows. Section 5.1 motivates the need for abstraction of ABL information from gate-level circuitry using a design for a multiplier with Booth-encoded partial products as an illustrating example. This is a typical example for designs with mixed ABL/gate-level representations. Section 5.2 details an approach to synthesize an ABL description from a gate netlist so that the full power of the normalization technique from Chapter 3 may further be used for ABL verification. Section 5.3 extends the concept of the ABL synthesis from Section 5.2 and presents an approach to compute normalized polynomials in a ring $\mathbb{Z}/2^n$ applicable for the computer algebra algorithms of Chapter 4. Section 5.4 provides experimental results for the ABL synthesis technique. The experimental data for the technique of Section 5.3 are collected separately. They are gathered in Chapter 6 where it is also shown how this technique can be integrated into the algebraic approach of Chapter 4 to solve SMT decision problems originated from formal verification of arithmetic designs.

5.1 Mixed ABL/Gate-Level Problems

As it pointed out in Chapter 3 and Chapter 4, the ABL-based approaches can successfully be applied whenever ABL information is available for the full RTL description of a design. Unfortunately, this is not always the case in industrial practice. In order to improve performance or area, designers, sometimes, describe certain parts of an arithmetic circuit below the ABL, i.e., at the gate level. In this case, the ABL-based approaches are not applicable.

For instance, the speed of computations performed by a multiplier is dominated by the additions of partial products. A widely adopted technique to reduce the number of partial products is (radix-4) Booth encoding [Kor98]. At the ABL, the Booth-encoded partial products can be described with the following equation:

$$p_i = (-2a_{(2i+1)} + a_{(2i)} + a_{(2i-1)}) \cdot b \cdot 2^{2i} = A \cdot b \cdot 2^{2i}, \tag{5.1}$$

where $A \in \{-2, -1, 0, 1, 2\}$ is a so-called Booth digit.

However, when implementing the logic of a Booth encoder, designers do not construct a signed $(3 \times n)$-bit multiplier and a $((3 + n) \times 2i)$-bit multiplier. Instead, they proceed as follows:

- multiplication with 2^j, where $j \in \mathbb{Z}_+$, is implemented by shifting the corresponding bit vector by j binary digits to the left,
- multiplication with a Booth digit $A \in \{-2, -1, 0, 1, 2\}$ can be implemented in the two following steps:
  - $*$ multiplication with $|A| = 2$ as an additional shift by one digit,
  - $*$ in case $A < 0$, the transformation $A \cdot b = \overline{-A \cdot b} + 1$ is used.

The conditions for the extra bit shift and the negation are defined as:

$$\mathit{shift}_i = (a_{(2i+1)} \wedge \overline{a_{(2i)}} \wedge \overline{a_{(2i-1)}}) \vee (\overline{a_{(2i+1)}} \wedge a_{(2i)} \wedge a_{(2i-1)}) \tag{5.2}$$

$$\mathit{cpl}_i = (a_{(2i+1)} \wedge (\overline{a_{(2i)}} \vee \overline{a_{(2i-1)}})) \tag{5.3}$$

Therefore, the implemented partial products $p'_i = (p_i - 2^{2i} \cdot \mathit{cpl}_i)$ can be computed as follows:

- $p'_i = 0$, if $a_{(2i+1)} = a_{(2i)} = a_{(2i-1)} = 0$,
- $p'_i = 0$, if $a_{(2i+1)} = a_{(2i)} = a_{(2i-1)} = 1$,
- $p'_i = ((b_{n-1} \oplus \mathit{cpl}_i, \ldots, b_0 \oplus \mathit{cpl}_i) \ll \mathit{shift}_i) \ll 2i$, otherwise.

Suppose, the addition of the partial products p'_i and the complement bits cpl_i is performed using a tree of carry-save adders (CSA tree). In this case, an ABL description is generated for the adder tree and for the standard multiplier implementation used in the property. However, the Booth-encoded partial products form a custom-designed component of the multiplier design. Therefore, they will not be a part of this ABL description. As a result, e.g., the normalization approach fails to identify equivalent partial products after merging the addition networks of the implementation and the specification, respectively. This problem is illustrated in Figure 5.1.

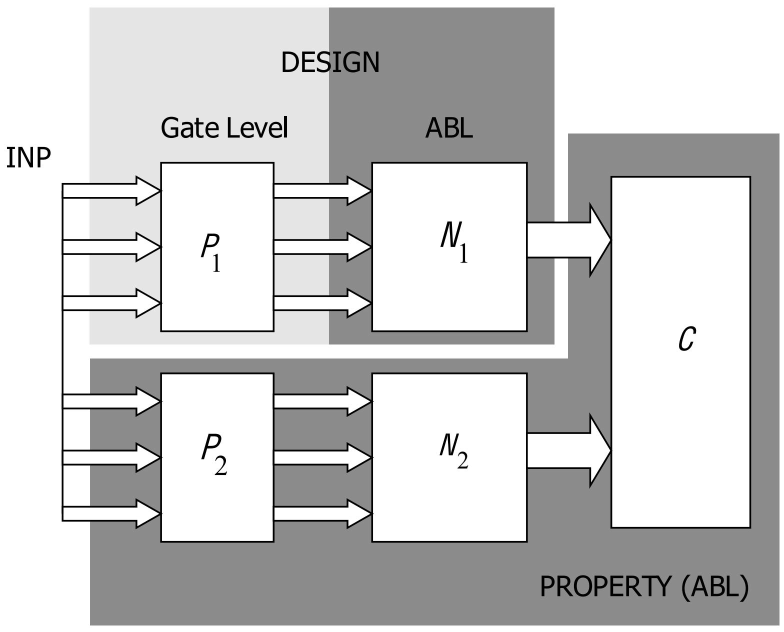


Figure 5.1: Incompletely normalized instance with a custom-designed component

In the next sections, we provide techniques to generate an ABL description for local custom-designed gate-level components of arithmetic circuits.

5.2 Modeling for ABL Normalization

This section describes a technique introduced in [PWS$^+$08] to transform a gate netlist into an ABL model applicable for the ABL normalization presented in Section 3.2.2.

5.2.1 Synthesis of ABL Descriptions from Gate Netlists

In principle, every Boolean function can be synthesized into an equivalent ABL description. Moreover, it is also possible to obtain an ABL description in reduced normal form. To see this, let us consider the Reed-Muller decomposition stated by Definition 2.10. Recursive application of this decomposition results in the Reed-Muller form for a given Boolean function. Sometimes, the Reed-Muller form is also called a Boolean polynomial (in sum-of-products notation).

Example 5.1. *(***Reed-Muller Forms***) Table 5.1 contains Boolean polynomials sketched for some logic primitives.*

Logic expression	$\overline{a}$	$a \wedge b$	$a \vee b$	$\overline{(a \wedge b)}$	$\overline{(a \vee b)}$
Reed-Muller form	$1 \oplus a$	$a \cdot b$	$a \oplus b \oplus (a \cdot b)$	$1 \oplus (a \cdot b)$	$1 \oplus a \oplus b \oplus (a \cdot b)$

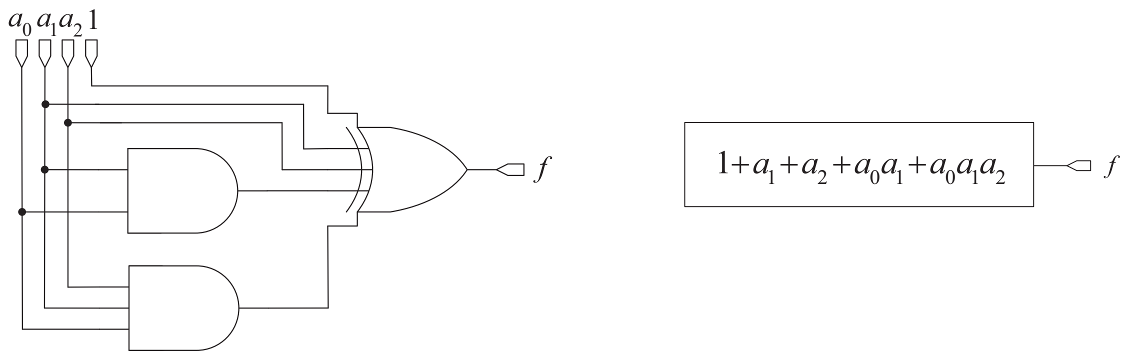
Table 5.1: Reed-Muller forms for some Boolean primitives

Suppose, a Boolean function is given in a positive Reed-Muller form. The product terms of the Reed-Muller form can be transformed into equivalent cascades of partial product generators, and the XOR function can be implemented by a single-column addition network. This case is illustrated with the next example.

Example 5.2. *(***Reed-Muller Form to a Single-Column Addition Network***) Recall the function $f = (a_0 \wedge a_1 \vee a_2) \oplus \overline{a}_1$ in Figure 2.3. According to the recursive application of the Reed-Muller decomposition from Definition 2.10, the Boolean polynomial for f is as follows:*

$$f = 1 \oplus a_1 \oplus a_2 \oplus a_0 a_1 \oplus a_0 a_1 a_2.$$

Figure 5.2 depicts the gate netlist and the single-column addition network for the Boolean polynomial of the function f.



(a) Gate netlist for Reed-Muller form (b) Single-column addition network

Figure 5.2: Transformation of Reed-Muller form to ABL model

However, by calculating also Reed-Muller forms from gate netlists of multi-column addition networks, the carry bits have to be taken into account. Moreover, the calculation of Reed-Muller forms for output bits of an arithmetic circuit is not feasible for circuits with larger bit width as soon as multiplication is involved. Fortunately, the Reed-Muller form can be built easily for smaller portions of such a circuit that may have been implemented in an optimized way at the gate level.

There are efficient data structures such as OFDDs [KSR92] and OKFDDs [DBS+94] to represent and manipulate Boolean functions in Reed-Muller form. As we only generate local Reed-Muller forms for small portions of a circuit, we do not resort to these data structures but, instead, represent Reed-Muller expressions explicitly. Starting from general logic gates, we generate the Reed-Muller form simply by local substitution of gates followed by application of the distributive law and elimination of duplicated product terms.

In the following section, we study how to transform the Reed-Muller form of a Boolean function into an ABL description that is suitable for the normalization approach reviewed in Section 3.2.2.

ABL Descriptions for Boolean Functions in Reed-Muller Form

As we have already stated above, we can implement any Boolean function in Reed-Muller form by means of an ABL description using appropriate partial product generators for the product terms together with a single-column addition network.

The addition network obtained in this way will always generate a carry in its single column, unless it consists of a single product term only. If the result of such a single-column addition network N with more than a single addend is used as an addend in some other network N', merging of N and N' is only possible if the result of N is added to the uppermost column of N'. This, however, cannot be expected in general since it would require that only addends to the uppermost column of an addition network implementation are specified at the gate level. In order to overcome this restriction, we extend a single-column addition network to an equivalent multi-colum addition network as shown and proved with the next theorem.

Theorem 5.1. *(***Multi-Column Extension of Addition Network***) Let N be a single-column addition network with result r. For every $n \geq 1$ there is a n-column addition network N' with results $(r'_{n-1}, \dots, r'_0)$ such that $r'_0 = r$ and $r'_i = 0$ for all $i > 0$.*

Proof: *Without loss of generality we can suppose N to have a set of addends $A = \{a_1, \dots, a_m\}$ with $w(a_i) = 1$ for all $a_i \in A$ and a constant offset $c \in \{0, 1\}$. We need to consider the case $c = 0$ only. Note that $c = 1$ can be handled by inserting a dummy addend a. In the resulting network N' we eliminate a by updating the constant offset to $c + w(a)$. In order to transform N, consider the n-column addition networks $\hat{N}$ with $\hat{r} = (\hat{r}_{n-1}, \dots, \hat{r}_0) = \langle (\sum_{i=1}^{m} a_i), n \rangle$ and $\tilde{N}$ with $\tilde{r} = (\tilde{r}_{n-1}, \dots, \tilde{r}_0) = \langle (\sum_{i=1}^{m} a_i - \sum_{i=1}^{n-1} 2^i \hat{r}_i), n \rangle$.*

Obviously, $\tilde{N}$ has the results $\tilde{r}_0 = r$ and $\tilde{r}_i = 0$ for all $i > 0$. Moreover, the $\hat{r}_i$ can be expressed as Boolean functions (in Reed-Muller form) in terms of the addends a_k. Therefore, we obtain a single-column addition network $\hat{N}_i (i > 0)$ for each of the $\hat{r}_i$. By induction hypothesis, we can extend $\hat{N}_i$ to an $(n - i)$-column addition network $\hat{N}'_i$. By construction, we can merge the addition networks $\hat{N}'_i$ with $\tilde{N}$ and obtain an equivalent network N'. □

By means of the above theorem, we can generate ABL descriptions for Boolean functions that are suitable for ABL normalization. This procedure is described in Example 5.3.

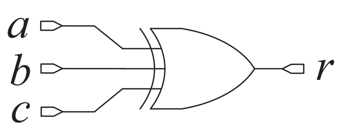


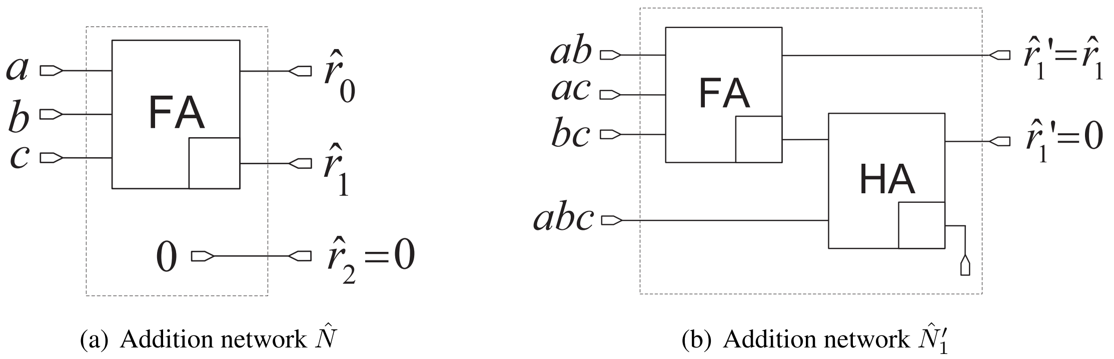
Figure 5.3: Reed-Muller form for $f = a \oplus b \oplus c$

Example 5.3. *(***Reed-Muller Form to a Multi-Column Addition Network***) Suppose we want to implement the XOR function of the variables a, b and c, as shown in Figure 5.3, by a three-column addition network N' with results $r' = (r'_2, r'_1, r'_0)$. Using the above variables a, b, c as addends in column 0, we obtain the Reed-Muller forms $\hat{r}_1 = ab \oplus ac \oplus bc$ and $\hat{r}_2 = 0$ for the results of the intermediate network introduced in the above proof, see the left side of Figure 5.4. As shown on the right side in Figure 5.4, $\hat{r}_1$ can be implemented by the two-column addition network $\hat{N}'_1$ with the addends ab, ac and bc in column 0 and the addend abc in column 1, where the result variables for this network are the following Reed-Muller forms $\hat{r}'_0 = ab \oplus ac \oplus bc$ and $\hat{r}'_1 = 0$. Note that the addend abc corresponds to the carry for the addition of ab, ac and bc. Merging $\hat{N}'_1$ with $\tilde{N}$ results in an addition network specified by the following equation:*

$$r' = \langle (a + b + c - 2(ab + ac + bc) + 4abc) \mod 8, 3\rangle$$

$$= \langle (a + b + c + 2(ab + ac + bc) + 4(abc + ab + ac + bc)) \mod 8, 3\rangle$$

for the result r' of the final addition network N'. This addition network can be implemented by the half-/full-adder netlist depicted in Figure 5.5.



(a) Addition network $\hat{N}$ (b) Addition network $\hat{N}'_1$

Figure 5.4: Intermediate ABL models

As a side effect, the above example also demonstrates how to eliminate negative weights for addends in column i. In this case, the corresponding addends are added to all columns $k \geq i$ of an addition network. This effect is explained more precisely by means of Example 5.4.

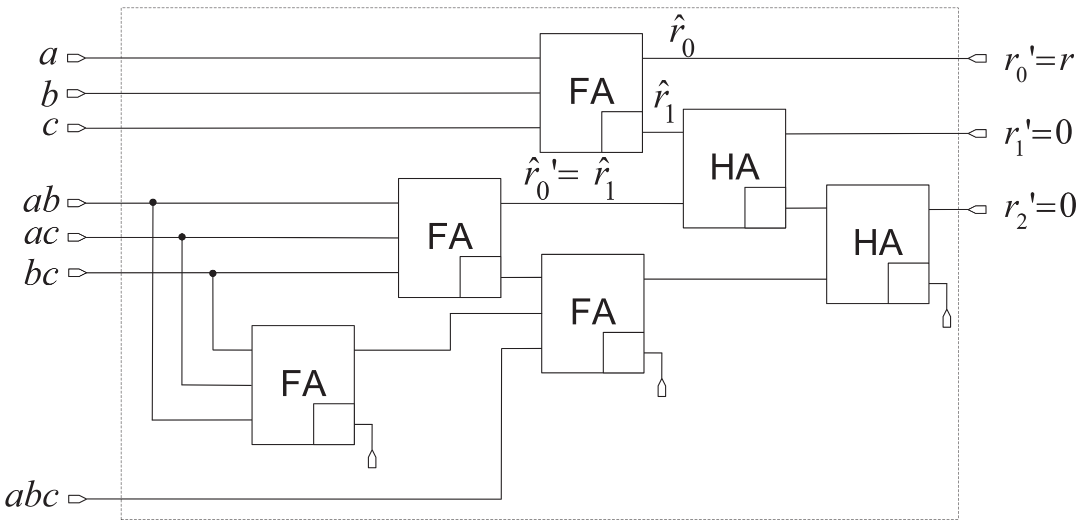


Figure 5.5: Addition network N' as a synthesized ABL model

	binary notation	decimal notation
1st operand	...111111111111	-1
2nd operand	...000000000001	+1
result	...000000000000	...000000000000

Table 5.2: Elimination of a negative weight in addition

Example 5.4. *(***Negative Weight Elimination***) Table 5.2 depicts the addition of two signed numbers, namely* (-1) *and* 1 *encoded in both binary and decimal representation. The binary value for the decimal number* (-1) *is nothing else but an infinite sequence of constant bits* 1*. On the contrary, the binary value for the decimal number* 1 *is a vector of constant bits* 0 *except the least significant bit where the constant bit* 1 *is placed. Thus, the addition of these two bit vectors will always produce a bit* 0 *as a result of sum for the current column and a bit* 1 *as a result of carry for the next column.*

Summarizing all studied above in this section, the overall procedure to synthesize ABL descriptions for local parts of the circuit can be implemented in two steps:

- transformation of local gate-level descriptions into Reed-Muller forms,
- transformation of the Reed-Muller forms into equivalent multi-column addition networks applicable for further ABL normalization.

In practice, this procedure is invoked on demand whenever conventional ABL normalization terminates with remaining addends in the compared addition networks. If some gate-level representations still exist in fanin to the compared addition networks then these representations are converted into ABL models. Afterwards, normalization is rerun.

The overall flow of this procedure is explained by means of the following example.

Example 5.5. *(***ABL Model for Booth-Encoder***) Suppose we want to verify the design of a 2x2 unsigned multiplier with (radix-4) Booth-encoded partial products like the multiplier design described in Figure A.6. We, further, assume that the partial products of the design are implemented at the gate level. The partial products provided for the addition tree of the design are listed in the left part of Table 5.3. Furthermore, we annotate the corresponding Reed-Muller forms in terms of the multiplier inputs* a_k *and* b_i*.*

column	result bit	Booth-encoded (radix-4) partial products	standard multiplier partial products
0	r_0	$cpl_0 = b_1$ $p'_0[0] = a_0b_0 \oplus b_1$	a_0b_0
1	r_1	$p'_0[1] = b_1 \oplus a_1b_0 \oplus a_0b_1 \oplus a_0b_0b_1$	a_1b_0, a_0b_1
2	r_2	$cpl_1 = 0$ $p'_0[2] = b_1 \oplus a_1b_1 \oplus a_1b_0b_1$ $p'_1[2] = a_0b_1$	a_1b_1
3	r_3	$\underline{signext} = 1$ $\overline{p'_0[3]} = \overline{cpl_0} = 1 \oplus b_1$ $p'_1[3] = a_1b_1$	

Table 5.3: Partial products for unsigned multipliers

Table 5.3 also corresponds to the addition network N_{Impl} obtained in the normalization algorithm after merging the adders in the addition tree of the implementation, i.e., the result of N_{Impl} is computed by an addition network for the following equation:

$$(r_3, r_2, r_1, r_0) = \langle(cpl_0 + p'_0[0] + 2p'_0[1] + 4(p'_1[2] + p'_0[2] + cpl_1) \\ +8(signext + p'_1[3] + 1 - p'_0[3])), 4\rangle.$$

The partial products for a standard implementation of a multiplier are depicted in right part of Table 5.3. It is obvious that normalization cannot establish equivalence between these addition networks, as the partial products a_0b_0 and a_1b_0 do not have an equivalent counterpart in the implementation network N_{Impl}.

In order to complete normalization, we convert the Reed-Muller forms of the partial products of the implementation into the corresponding multi-column addition networks. The result of this computation step is summarized in Table 5.4.

partial product	Reed-Muller form	addition network
cpl_0	b_1	$(0,0,0,cpl_0) = \langle b_1, 4\rangle$
$p'_0[0]$	$a_0b_0 \oplus b_1$	$(0,0,0,p'_0[0]) = \langle(a_0b_0 + b_1 - 2a_0b_0b_1), 4\rangle$
$p'_0[1]$	$b_1 \oplus a_1b_0 \oplus a_0b_1 \oplus a_0b_0b_1$	$(0,0,p'_0[1]) = \langle(b_1 + a_1b_0 + a_0b_1 + a_0b_0b_1$ $-2(a_1b_0b_1 + a_0b_1), 3\rangle$
$p'_1[2]$	a_0b_1	$(0,p'_1[2]) = \langle a_0b_1, 2\rangle$
$p'_0[2]$	$b_1 \oplus a_1b_1 \oplus a_1b_0b_1$	$(0,p'_0[2]) = \langle(b_1 + a_1b_1 + a_1b_0b_1 - 2a_1b_1), 2\rangle$
cpl_1	0	$(0,cpl_1) = \langle 0, 2\rangle$
signext	1	$(signext) = \langle 1, 1\rangle$
$p'_1[3]$	a_1b_1	$(p'_1[3]) = \langle a_1b_1, 1\rangle$
$\overline{p'_0[3]}$	$b_1 \oplus 1$	$(\overline{p'_0[3]}) = \langle 1 + b_1, 1\rangle$

Table 5.4: Multi-column addition network for implementation products

Merging the addition networks of Table 5.4 for the partial products with the addition network N_{Impl} of the implementation results in the addition network N'_{Impl} with

$$(r_3, r_2, r_1, r_0) = \langle(b_1 + (a_0b_0 + b_1 - 2a_0b_0b_1) + 2(b_1 + a_1b_0 + a_0b_1 + a_0b_0b_1 - 2(a_1b_0b_1 + a_0b_1)) \\ +4(a_0b_1 + (b_1 + a_1b_1 + a_1b_0b_1 - 2a_1b_1) + 0) + 8(1 + a_1b_1 + 1 + b_1)), 4\rangle \\ = \langle(16 + 16b_1 + a_0b_0 + 2a_1b_0 + 2a_0b_1 + 4a_1b_1)), 4\rangle = \langle(a_0b_0 + 2a_1b_0 + 2a_0b_1 + 4a_1b_1)), 4\rangle$$

Obviously, the resulting addition network N'_{Impl} and the standard addition network for unsigned multiplication are identical. Therefore, our implementation is proven to be correct.

The approach presented above in this section was used as a preprocessor for ABL normalization technique [WSBK07]. The modified version of this technique was successfully tested with more than 1000 different examples instantiating arithmetic verification problems. The detailed report on these experiments is given in Section 5.4.

5.3 Modeling for Algebraic Approach

In this section, we adapt the extraction techniques described in Section 5.2 for deriving compact polynomial representations in a ring $\mathbb{Z}/2^n$ for those parts of the arithmetic decision problems that are defined using Boolean constraints of the bit-vector logic. We reformulate this extraction technique as a combination of arithmetic transform computation and a so-called pre-normalization so that the algebraic approach [PWS$^+$11] presented in Chapter 4 may further be applied.

5.3.1 Extraction of Arithmetic Bit Level Information

In order to treat non-arithmetic constraints with the algebraic approach discussed in Chapter 4, we need to generate a set of polynomials $\mathbb{Z}/2^n$ that captures the variable relations specified by these constraints. We illustrate the extraction process using a custom-designed circuit component presented in the next example.

Example 5.6. *(**Adder**) Figure 5.6 depicts a gate netlist of a two-bit adder which may be a possible custom-designed component for the RTL design of an arithmetic circuit.*

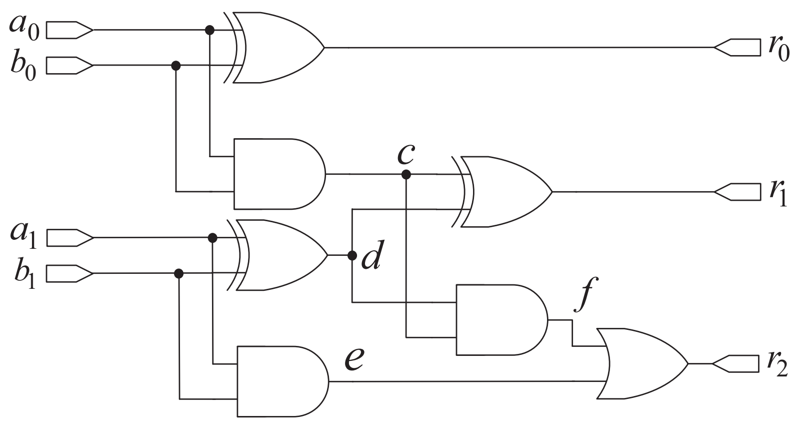


Figure 5.6: Gate netlist of a two-bit adder

Note that the front-end of a standard property checker, usually, performs a one-to-one compilation of such a circuit structure into a collection of Boolean bit-level constraints of the bit vector logic. We consider a set of Boolean constraints that stem from a connected part of the bit-vector netlists representing an SMT instance. By a topological analysis, we may identify the variables x_j used as inputs and variables y_i used as outputs of this sub-netlist under consideration. Therefore, we propose two phases for our technique to extract polynomials as follows:

- derive polynomial equations based on the arithmetic transform for each Boolean constraint,
- pre-normalize the polynomials with respect to the input variables x_j.

In Table 5.5, we summarize the polynomial equations for a representative subset of Boolean constraints that may be used in a gate netlist.

Boolean domain	ABL domain
$o = x \otimes y$	$o = (x + y - 2xy)$ mod 2
$o = x \wedge y$	$o = (xy)$ mod 2
$o = x \vee y$	$o = (x + y - xy)$ mod 2
$o = \overline{x}$	$o = (1 - x)$ mod 2
where $x, y \in \mathbb{B}$	

Table 5.5: Polynomial equations based on the arithmetic transform of Boolean gates

For Example 5.6, the first step of the proposed technique results in the set of equations illustrated in Equation 5.4. In theory, these equations may immediately be used as starting point for our algebraic models generated in Chapter 4.

$$\begin{cases} r_0 = (a_0 + b_0 - 2a_0b_0) \text{ mod } 2 \\ c = (a_0b_0) \text{ mod } 2 \\ d = (a_1 + b_1 - 2a_1b_1) \text{ mod } 2 \\ r_1 = (c + d - 2cd) \text{ mod } 2 \\ f = (cd) \text{ mod } 2 \\ e = (a_1b_1) \text{ mod } 2 \\ r_2 = (e + f - ef) \text{ mod } 2 \end{cases} \tag{5.4}$$

However, it is already apparent from the example that this fine-grained modeling of the non-arithmetic constraints in the cone of influence of an algebraic proof goal will lead to a fairly large problem instance if multiple such problem parts exist. For instance, the two bit adder of Example 5.6 may many times be instantiated within a large multiplier. This unnecessary blowup, both in terms of the numbers of variables as well as in terms of the numbers of polynomials, can considerably slow down the normal form computation. Fortunately, the custom-designed components, frequently used within such designs, have a compact polynomial representation, e.g., as demonstrated with the next example.

Example 5.7. (**Compact Representation**) *Equation 5.5 provides a compact description of the input-/output behavior for the two bit adder from Example 5.6.*

$$4r_2 + 2r_1 + r_0 = 2(a_1 + b_1) + (a_0 + b_0) \; mod \; 8 \tag{5.5}$$

In our extraction technique, we perform a pre-normalization in order to achieve such a compact representation. This step will be described in the remainder of this section.

We start with the polynomial set P generated from the above equations using the models of Chapter 4. We consider the subset $O \subset P$ of polynomials that depend on the output variables y_i. Recall that we may assume a monomial order that is compatible with the topological order of the problem netlist. In this case, the polynomials in O have a variable y_i as leading monomial.

For each polynomial $f \in O$, we compute the reduced normal form with respect to the polynomials in $P \backslash O$. Note that the polynomials $g \in P$ used during this computation for the reduction of the polynomials f or, more precisely, for the reduction of the tails of the polynomials f can be determined efficiently by backward tracing the problem netlist at hand. Let us illustrate this process with the example below.

Example 5.8. *(***Reduced Normal Form Computation***) For the circuit from Example 5.6, let us consider the output variables r_0, r_1, r_2, first, and, then, traverse the circuit backwards until primary input variables a_0, b_0, a_1, b_1 are reached. All polynomials during this traversal are immediately used to reduce the polynomials for the r_i.*

For example, for the output r_1, this tracing procedure delivers the following set of polynomials:

$$\begin{cases} f = r_1 - (c + d - 2cd) \\ g_1 = c - (a_0 b_0) \\ g_2 = d - (a_1 + b_1 - 2a_1 b_1) \end{cases} \tag{5.6}$$

Tail reduction of f with respect to g_1 and g_2 results in the polynomial:

$$r_1 - (a_1 + b_1 + a_0 b_0 - 2a_1 b_1 - 2a_0 a_1 b_0 - 2a_0 b_0 b_1 + 4a_0 a_1 b_0 b_1).$$

Equation 5.7 lists the complete set of polynomials obtained for the example from Figure 5.6.

$$\begin{cases} f_0 = r_0 - (a_0 + b_0 - 2a_0 b_0) \\ f_1 = r_1 - (a_1 + b_1 + a_0 b_0 - 2a_1 b_1 - 2a_0 a_1 b_0 - 2a_0 b_0 b_1 + 4a_0 a_1 b_0 b_1) \\ f_2 = r_2 - (a_1 b_1 + a_0 a_1 b_0 + a_0 b_0 b_1 - 2a_0 a_1 b_0 b_1) \end{cases} \tag{5.7}$$

These pre-normalized polynomials are, then, used to model the non algebraic problem part for the normal form computation in Chapter 4.

To conclude this example, it should be noted that the weighted addition $4f_2 + 2f_1 + f_0$ of the above polynomials yields the same compact polynomial

$$4r_2 + 2r_1 + r_0 - ((a_0 + b_0) + 2(a_1 + b_1))$$

as it was defined with Equation 5.5.

The ABL modeling approach of this section will be used as a substantial part in an RTL-/SMT solving engine described and examined in Chapter 6.

5.4 Experimental Results

In this section, the results of the experimental evaluation are summarized for the techniques proposed in Section 5.2. These techniques were implemented in a property checking environment utilizing SAT and ABL normalization [WSBK07]. The overall flow of the integrated verification engine is as shown in Figure 5.7.

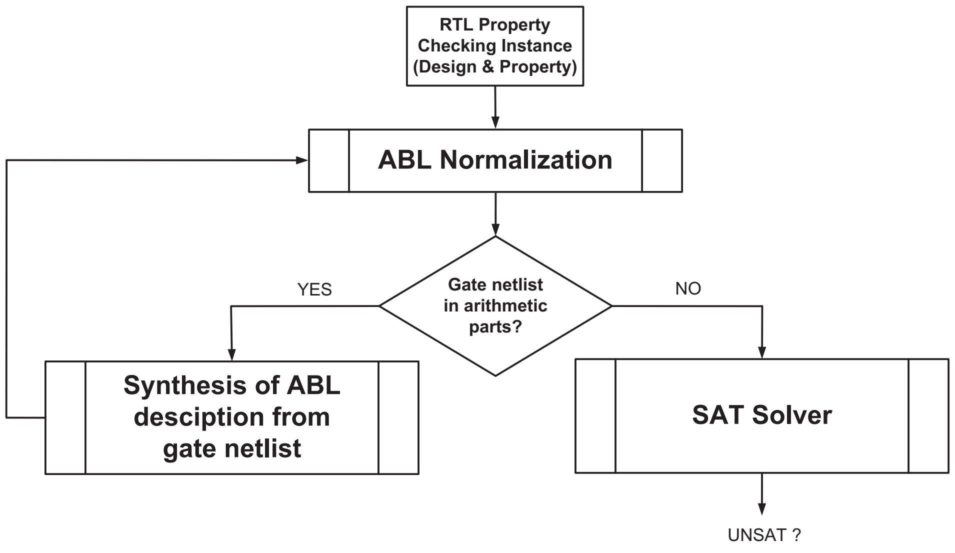

RTL Property Checking Instance (Design & Property)
ABL Normalization
Gate netlist in arithmetic parts?
YES
NO
Synthesis of ABL desciption from gate netlist
SAT Solver
UNSAT ?

Figure 5.7: Flow of RTL property checking used in experiments

This verification engine expects a combinatorial netlist as an input instance. Such a netlist can be derived from an HDL design and a property using the industrial property checker OneSpin 360 MV [One].

In order to simplify the corresponding SAT instance, it is necessary to normalize an ABL description generated from the arithmetic bit vector functions in this netlist. This may result in proven equivalences between arithmetic signals of the design and their specification given in the property. However, if certain parts of the arithmetic circuit design are implemented at the gate level, normalization will not succeed. In this case, such non-arithmetic bit vector functions are determined in the fanin of design signals that are still present in the normalized ABL description. For these bit vector functions, equivalent ABL representations are derived and included into the ABL normalization problem.

The process of extending the ABL description followed by normalization is iterated until either all comparisons for arithmetic signals are proven or no suitable extension of the normalized ABL description can be generated. In both cases, a SAT solver is called to prove unsatisfiability of the remaining parts of the problem.

The performance of the prototype implementation was compared against the following state-of-the-art SMT solvers: *Spear-2-7* [BH08, Spe], *Boolector 1.4* [BB09, boo], *MathSAT v. 4.3-smtcomp* [BBC+05], and *simplifyingSTP* based on revision 939 of *STP* [GD07, STP]. It should be noted that these solvers or their earlier versions were winners at the SMT competitions 2007-2010 [SMT07, SMT08, SMT09, SMT10].

All experiments reported in this section were carried out on Intel Xeon CPU E5420 2,5 Ghz 32 GB RAM running Linux. The value of the memory limit (*MO*) was set to 8 GB for each instance. Also, the run time for each instance was limited. The time-out limits (*TO*) are specified in the captions for the tables and for the figures with experimental data presented below in this section. The symbol *U* denotes a case when a tool was not able to treat an instance properly and terminated with the result *"unknown"*. This, usually, happens because of some internal errors like, e.g., incorrect parsing or erroneous memory deallocation.

5.4.1 Industrial Multiplication Design Benchmark

As a first step of the evaluation, experimental data were collected in an industrial setting. Different multiplier implementations with input bit width in the range from 4 bits to 64 bits were generated by the module generator of an industrial synthesis tool. The total suite of all instances consists of 1040 implementations. The RTL code of the synthesized circuits includes Booth-encoded partial products described at the gate level and CSA trees implementing the addition network at the ABL.

Selected experimental results for signed and unsigned Booth-encoded multipliers of various bit widths are collected in Table 5.6. A complete view of the experimental results is depicted in Figure 5.8. This figure represents plots with quotas of instances as they were solved by the refined ABL normalization technique in comparison to contemporary SMT solvers. The experimental data for bounds of quotas are shown in Table 5.7.

The experimental results indicate that only SMT solvers used for comparison are adequate to prove the arithmetic properties for small instances. Due to limited computing resources, they abort for instances of realistic size. However, when the missing ABL blocks are added to the arithmetic proof problem, the ABL normalization-based tool can perform the proof of correctness within less than one minute.

multiplier design	verification approach				
	Spear	Boolector	MathSAT	simplifyingSTP	ABL
signed 8x8	99.4	13.4	32.14	13.87	0.06
signed 16x16	TO	TO	TO	TO	0.44
signed 23x23	TO	TO	TO	TO	1.4
signed 32x32	TO	TO	TO	TO	2.57
signed 64x64	TO	TO	TO	TO	45
unsigned 8x8	46.14	14.9	21.13	16.65	0.08
unsigned 16x16	TO	TO	TO	TO	0.52
unsigned 23x23	TO	TO	TO	TO	1.4
unsigned 32x32	TO	TO	TO	TO	3
unsigned 64x64	TO	TO	TO	TO	38.6

Table 5.6: Industrial multipliers / CPU times, sec, / TO: 1000 sec.

5.4.2 Limitation of Reed-Muller-based Extraction

The second step of the evaluation explores the capacity limits of the Reed-Muller extractor. There were generated unsigned multipliers for different input bit-widths and different

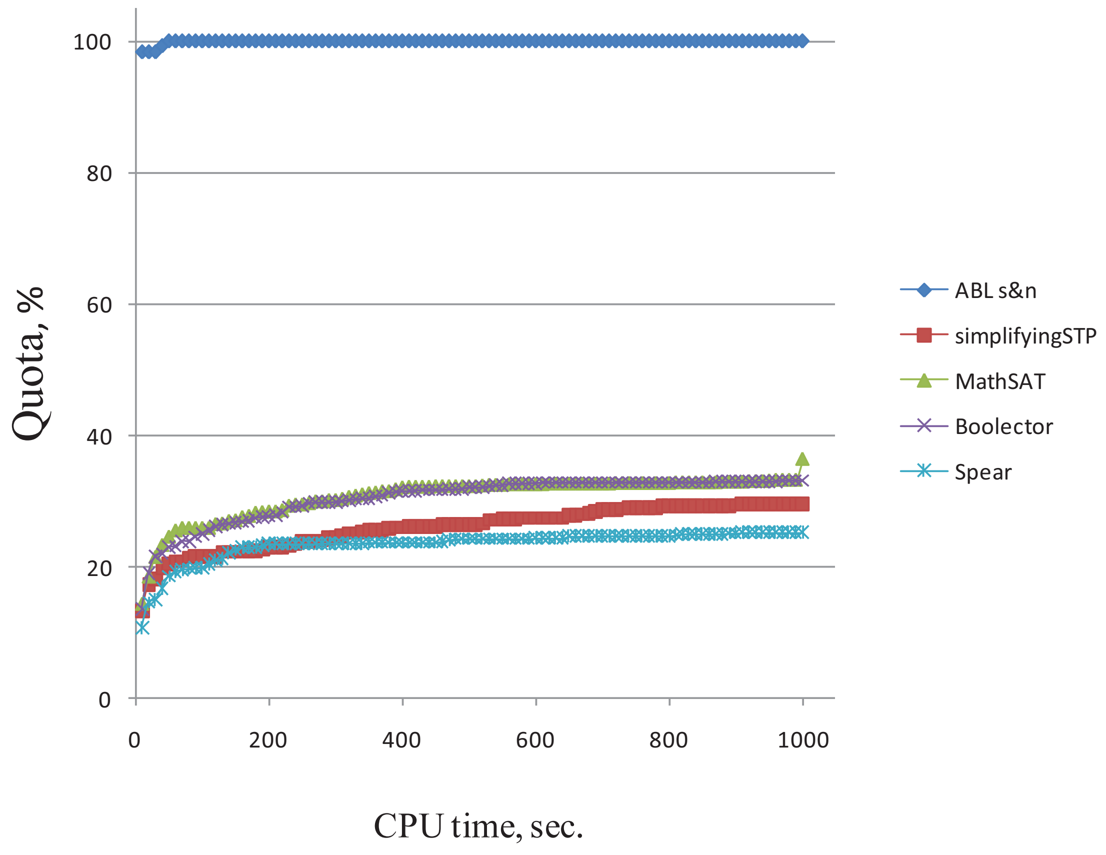


Figure 5.8: Quota of combinatorial instances derived from industrial multiplication designs for property checking and solved by different approaches / TO - 1000 sec.

CPU times, sec.	Spear	Boolector	MathSAT	simplifyingSTP	ABL
< 10	10.8%	13.7%	14.4%	13.4%	98.46%
< 100	19.9%	25.1%	25.9%	21.6%	100%
< 250	23.6%	29.3%	29.5%	23.8%	100%
< 500	24.3%	32.1%	32.3%	26.5%	100%
< 1000	25.3%	33.2%	36.4%	29.5%	100%
TO/MO/U	74.7%	66.8%	63.6%	70.5%	0%

Table 5.7: Quota of industrial multiplier instances solved by different approaches

encodings for partial products. The size of the design portion specified at the gate level was increased step-wise. The ABL part of the design was reduced at the same time. Starting with an initial design where only the partial products are specified at the gate level, the CSAs were iteratively implemented within the addition network by gate-level components. Figure 5.9 visualizes the first two iterations of this experiment. This experimental setup will end up with instances where the complete multiplier is specified below the ABL abstraction and, of course, it is not expected that a technique based on the Reed-Muller form may become efficient in this extreme case.

The results of Table 5.8 show that the proposed techniques are capable of transforming fairly large regions of circuitry specified at the logic level into equivalent ABL descriptions. In particular, if the borderline between the (optimized) partial product generators and the addition network is blurred – as is often the case in custom-designed circuits – the proposed approach is powerful enough to transform the complete partial product generator as well as at least the first level of addition logic into ABL.

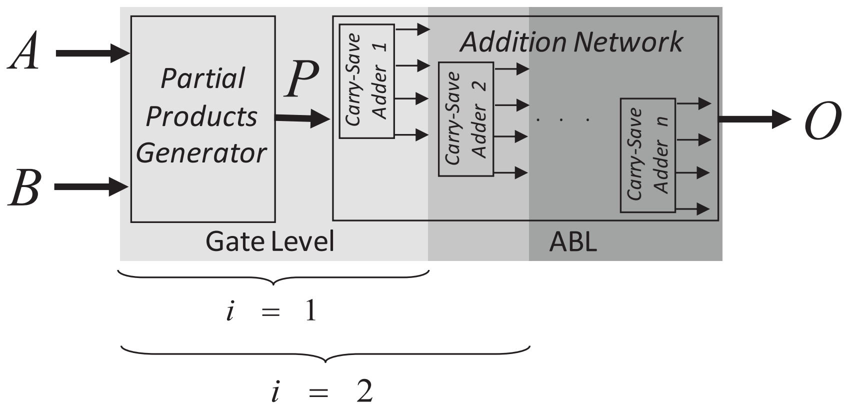


Figure 5.9: Increasing amount of the gate netlist with one carry-save adder per each iteration i, where $0 < i \leq n$ and n is a number of all carry-save adders in the circuit

unsigned multiplier design	standard multiplication scheme					Booth-encoded multiplication scheme	
	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 1$	$i = 2$
16x16	0.4	0.42	1.8	165	TO	38.7	TO
24x24	1.86	2	4.4	353	TO	72	TO
32x32	4.9	5.2	8.3	548	TO	103	TO
64x64	143	149	152	TO	TO	292	TO

Table 5.8: Multipliers of Figure 5.9 / CPU times, sec. / TO: 1000 sec.

5.4.3 Shared Multiplier Design Benchmark

To conclude the experimental evaluation of the proposed techniques, experiments were conducted with a sequential implementation for multiplication of four input values. In order to perform this operation with a single multiplier, it is used for the three required multiplication operations within three subsequent clock cycles. A finite state machine controls the assignment of the operands to the inputs of the multiplier as depicted in Figure 5.10. The figure also illustrates that the instantiated multiplier consists of a Booth encoder for the partial products implemented at the gate level and an addition network implemented at the ABL. The investigated property proves that every four clock cycles a correct arithmetic result is generated, provided that the reset and enable signals are triggered by the environment accordingly.

The resulting decision problem after unrolling the design and assumption propagation is depicted in Figure 5.11. Note that assumption propagation is used to eliminate control logic from the unrolled problem instance such that the remaining problem can be solved with the ABL normalization technique described in Chapter 3.

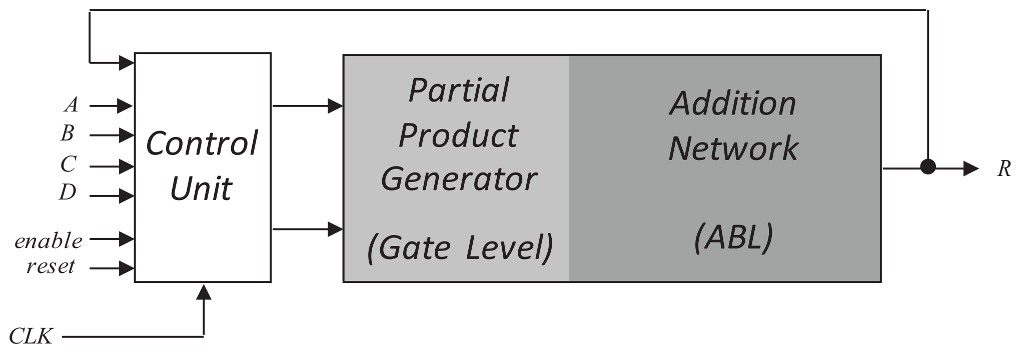


Figure 5.10: Shared multiplier

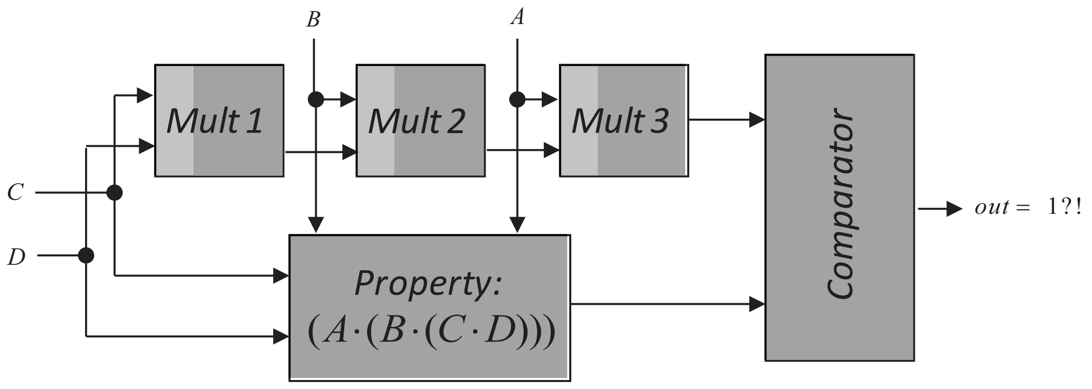


Figure 5.11: Property checking instance

The experimental results for different operand bit widths are summarized in Table 5.9. Again, the ABL-based tool demonstrates good performance and solves the verification task within reasonable time even for large instances.

input operand	verification approach				
bit-width	Spear	Boolector	MathSAT	simplifyingSTP	ABL
10	TO	TO	TO	TO	26
12	TO	TO	TO	TO	75
14	TO	TO	TO	TO	198
16	TO	TO	TO	TO	509
18	TO	TO	TO	TO	1002
20	TO	TO	TO	TO	2366
22	TO	TO	TO	TO	TO

Table 5.9: Shared multipliers of Figures 5.10, 5.11/ CPU times, sec. / TO - 3600 sec.

Chapter 6

STABLE: a new SMT Solver

This chapter describes a new SMT solver *STABLE*. This solver combines two recently developed techniques, namely the computer-algebra-based approach studied in Chapter 4 and the extraction method discussed in Chapter 5. STABLE was initially presented in [WPD$^+$10, PWS$^+$11] and designed to solve formulas of the quantifier-free logic over fixed-sized bit vectors (QF-BV).

As primary application domain for the proposed SMT solver, we will target a property checking flow for *System-on-Chip (SoC)* module designs. When verifying data-path modules for high performance applications, verification engineers frequently face custom-designed arithmetic components that are specified at the logic level of the respective hardware description language. This results in verification problems where arithmetic problem parts may include non-arithmetic constraints. Since the proposed verification technique includes the algorithm to extract arithmetic bit level information for these non-arithmetic constraints, the integrated algebraic approach becomes applicable to completely solve arithmetic subproblems. The non-arithmetic problem parts together with the results of the computer algebra engine are finally bit-blasted and handled by a standard SAT-solver.

This chapter is organized as follows. Section 6.1 provides description of the overall flow for the proposed SMT solver. Section 6.2 reports on experimental results. The evaluation of STABLE is shown in comparison to other contemporary SMT solvers on a large collection of SMT formulas describing verification problems of industrial data-path designs that include multiplication.

6.1 QF-BV SMT Solving

This section outlines a new SMT solver STABLE [WPD$^+$10, PWS$^+$11] for solving hard QF-BV decision problems that originate from verification of data paths that include multiplier components. The basic design flow of STABLE is illustrated in Figure 6.1 and explained below in this section.

The SMT solver integrates the algorithms studied in Section 2.7 and Section 5.3 for solving arithmetic subproblems of an input instance in question. A standard SAT-solver is used as a back end for solving the non-arithmetic problem parts of the input instance.

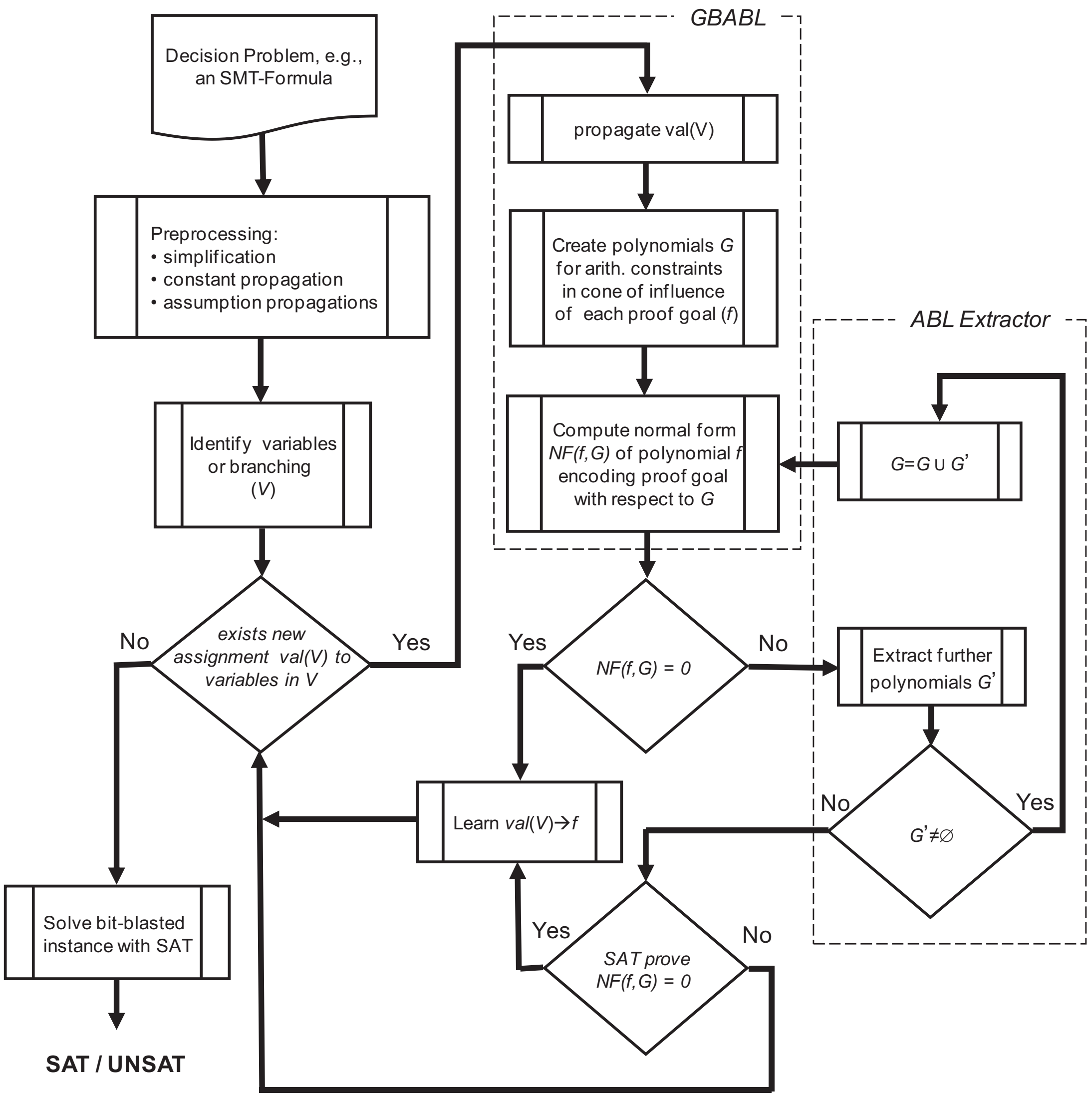

GBABL
Decision Problem, e.g., an SMT-Formula
Preprocessing:
• simplification
• constant propagation
• assumption propagations
Identify variables or branching (V)
exists new assignment val(V) to variables in V
No
Yes
propagate val(V)
Create polynomials G for arith. constraints in cone of influence of each proof goal (f)
Compute normal form NF(f,G) of polynomial f encoding proof goal with respect to G
ABL Extractor
G=G ∪ G'
NF(f,G) = 0
Yes
No
Extract further polynomials G'
G'≠∅
No
Yes
Learn val(V)→f
SAT prove NF(f,G) = 0
Yes
No
Solve bit-blasted instance with SAT
SAT / UNSAT

Figure 6.1: Flowchart of SMT Solver STABLE for solving QF-BV decision problems with *GBABL (Gröbner basis-based ABL) engine* and *ABL extractor*

This SAT solver operates on a bit-blasted version of the instance and also serves as a last resort in cases where the above mentioned techniques cannot fully solve an arithmetic subproblem.

In Figure 6.1, the dashed boxes indicate the interface between the computer algebra library (*GBABL* engine), the extraction library (*ABL extractor*), and the solver infrastructure.

As usual, the SMT solver performs a certain amount of preprocessing as a first step after parsing the SMT instance. Internally, the instance is represented as a netlist of predefined bit-vector functions. Since interval property checking is the primary application domain for the proposed technique, one may often face formulas with an implicative structure $(a \rightarrow c)$. In such formulas, a is a conjunction of assumptions that, e.g., may describe a scenario in which a property is applicable. Likewise, c is a conjunction of proof goals, also referred to as commitments, that have to be proven under these assumptions.

The SMT-LIB format [RT06, BST10] allows to represent the instances as a collection of constraints that are implicitly conjoined via the use of common variables. In this case, a topological analysis of the instances is employed to restore the netlist.

Given the bit-vector netlist, we start to treat the netlist by asserting the assumption $a = 1$. Usually, this assignment can be propagated into the netlist and can, thus, be used to simplify the proof goals in c. In problems for data-path verification, for example, such a propagation may eliminate control logic from the data path if the assumption implies a certain configuration of the data path. This may greatly simplify the solution process. In the optimal case when a unique configuration of the data path can be identified by propagation, a single pass through the main loop, that analyzes each of the configurations, is required.

The configurations to be analyzed are determined in the first step after preprocessing. More precisely, a set of Boolean branching variables V is determined that influence arithmetic subproblems in c. As branching variables we will consider the select inputs s used as condition in if-then-else constraints. Note that a careful selection of these variables is crucial as it is desirable to keep their number small. By topological analysis, we may omit many if-then-else constraints that do not influence arithmetic proof goals.

Given the set V of branching variables, STABLE enumerates each possible assignment $val(V)$. Note that in case of an empty set V, the loop is entered exactly once. After constant propagation of the assignment $val(V)$, STABLE analyzes the arithmetic proof goal f relevant under the respective configuration of the data path with a normal form computation. Note that an instance with multiple arithmetic proof goals may require an iteration of these two steps for each of the proof goals. For brevity, this extra loop is omitted in the flow in Figure 6.1.

To analyze a proof goal f, the proposed SMT solver generates a set G of polynomials $G \subset \mathbb{Z}/2^n$ modeling the arithmetic constraints in the cone of influence of f. Next, the *GBABL* engine computes the normal form (NF) of f with respect to G. If this normal form is the zero polynomial, STABLE learns the validity of the proof goal f under the current data-path configuration $val(V)$ and proceeds with the next iteration of the main loop.

However, it is also necessary to consider the case where the normal form of the proof goal is a non-zero polynomial. This may have the following reasons:

- some portions of the arithmetic problem part under consideration are expressed in terms of non-arithmetic constraints, e.g., they may originate from a custom-designed component in the data path of an RTL design,
- Boolean propagation of the assumptions and the assignment to branching variables $val(V)$ may not have been strong enough to completely eliminate the control logic separating parts of the data path,
- the proof goal is indeed invalid because it may stem from an invalid property that detects a bug in the design under verification.

STABLE approaches these issues as follows. First, it analyzes the normal form of the proof goal NF(f) whether one may extract additional polynomials that determine some of the remaining variables in the normal form of the proof goal. The resulting polynomials G' will have these variables as leading monomials. The SMT solver unifies the resulting polynomials set G' with the original set G and continues normal form computation.

In case the extraction process can not identify further polynomials, STABLE conducts a resource limited SAT check whether the normal form vanishes under the assumptions of the surrounding input instance. The SMT solver temporarily creates a bit-blasted version of this check for the SAT engine. Here, the incremental SAT interface of the underlying solver is used. In a current configuration of an SMT solver, the CPU time spent on each of these checks should be limited in a proper way. For example, in STABLE the value for this CPU time limit is set to 2 minutes. Note that the resource limitation is crucial. Some intermediate proof goals may be extremely hard to proof by SAT before other proof goals have been proven by the normal form engine.

In case when STABLE succeeds in proving that the normal form is equal to zero then the SMT solver may learn the constraint $val(V) \rightarrow f$. Note that in the frequent special case when V is empty, i.e., one needs to consider a single configuration of the data path, this implication only consists of the unit clause f. In this special case, STABLE also performs an additional propagation step that may considerably simplify other proof goals as well.

When all configurations of the data path, that are consistent with the assumptions of the overall input instance, have been analyzed, i.e., all valuations of V have been enumerated, STABLE again bit-blasts the entire instance including the learned constraints and the corresponding propagation results. Then, the SAT check, conducted on the resulting instance, may prove non-arithmetic proof goals as well as those proof goals where the strongly resource limited checks inside the loop have been aborted. As a result of this SAT check, a satisfying assignment is obtained for the input SMT formula or the formula is proven to be unsatisfiable.

6.2 Experiments

This section provides the results of an extensive evaluation of STABLE. Similar to the experiments reported in Section 5.4, the performance of STABLE is compared against four alternative state-of-the-art solvers, namely *Spear-2-7* [BH08, Spe], *Boolector 1.4* [BB09, boo], *MathSAT v. 4.3-smtcomp* [BBC$^+$05], and *simplifyingSTP* based on revision 939 of *STP* [GD07, STP]. These solvers or their predecessors have proven to be effective for the logic QF-BV, more precisely, they won the SMT competitions 2007-2010 in this category [SMT07,SMT08,SMT09,SMT10]. Additionally to that, two other well-known SMT QF-BV solvers – *Z3 v. 2.18* [Z3,MB08] and *Yices v. 2.0* [Yic,DM06,DdM06] – were also used for the experimental evaluations.

All experiments reported in this section were carried out on Intel Xeon CPU E5420 2,5 Ghz 32 GB RAM running Linux with a time-out limit (*TO*) of 1000 sec. and a memory limit (*MO*) of 8 GB per instance. Besides the standard reasons for aborting a respective instance, some instances are indicated as *unknown* if the solver crashes or aborts with a result other than SAT/UNSAT. For instance, *Spear* frequently terminates with a failing software assertion as soon as the bit width of a variable exceeds 64 bits inside an SMT instance. Thus, the abbreviation *MO/TO/U* summarizes three possible reasons for aborting a respective instance in all subsequent tables depicted in the next sections of this chapter.

The experiments are categorized into a few groups in accordance to the used benchmarks as described in Sections 6.2.1 to 6.2.4.

6.2.1 SMT Competition 2009 Benchmark

The evaluation was started by considering instances of the benchmarks of the 2009 SMT competition [SMT09]. As many of these benchmarks do not contain complex arithmetic subproblems with multiplication, it was expected that the specific strength of STABLE for arithmetic verification may not become visible. This is confirmed by Table 6.1 that lists quotas of instances solved per time-out limit. Obviously none of solvers were able to solve the complete suite of all formulas. A more detailed graphical demonstration of quotas for individual solvers is illustrated by the plots in Figure 6.2.

As expected, STABLE performs slightly slower than its competitors on these bench-

CPU time, sec.	Spear	Boolector	MathSAT	simplifyingSTP	STABLE
< 10	82,3%	91,4%	97%	88,9%	83,9%
< 100	89,4%	97,5%	98,5%	97%	95%
< 250	89,9%	98,5%	99%	98,5%	96,5%
< 500	90,4%	98,5%	99%	98,5%	96,5%
< 1000	90,9%	98,5%	99%	99%	97%
MO/TO/U	9,1%	1,5%	1%	1%	3%

Table 6.1: Quota of SMT-COMP09 instances solved by different SMT solvers

marks. This is mainly due to the fact that many instances contain non-arithmetic problems. Moreover, STABLE does not yet provide advanced rewriting mechanisms for the non-arithmetic problem parts. Nonetheless, STABLE is competitive. Besides a few exceptions, the run time for STABLE and its competitors differs by less then 200 seconds per instance.

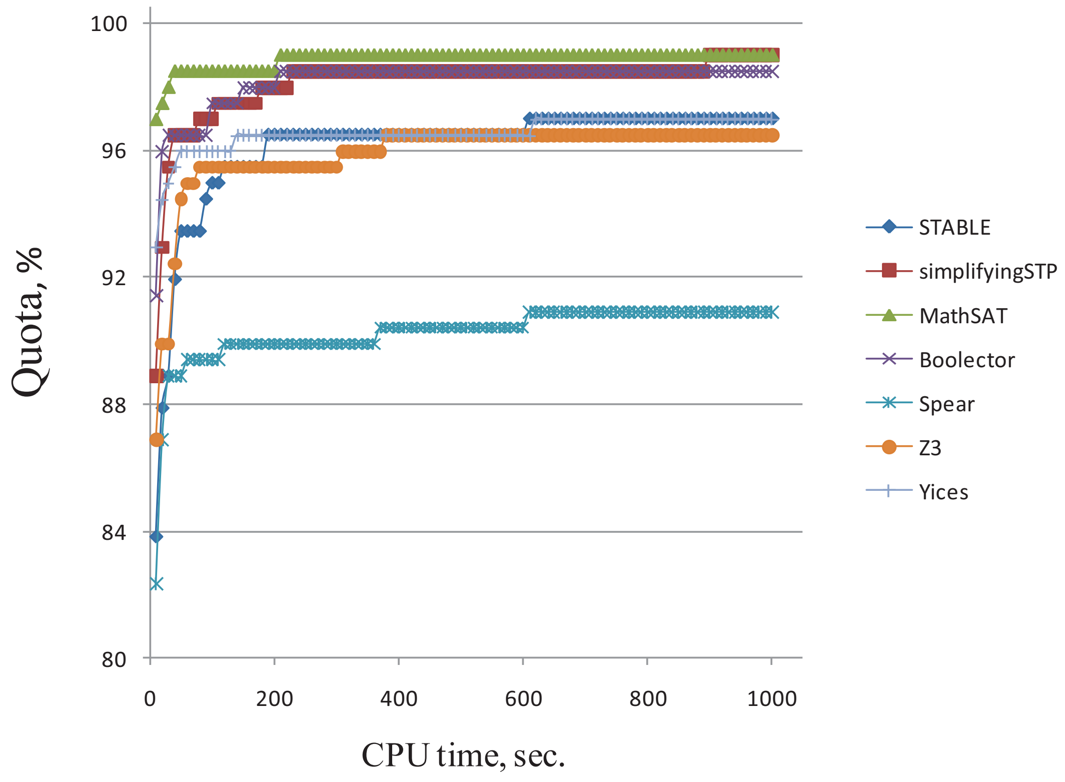


Figure 6.2: Quota of QF-BV SMT formulas from the suite of SMT competition 2009 solved by STABLE and its competitors / TO - 1000 sec.

6.2.2 Module Generator Benchmark

To evaluate the particular strength of STABLE in its primary application domain, experiments were conducted with two extensive sets of benchmarks originating from industrial verification projects for data paths that include multiplication. The first class of benchmarks is examined in this section whereas the second class is considered in Section 6.2.3.

The first class of benchmarks proves functional correctness for the outcome of the module-generator of an industrial synthesis tool. The resulting instances contain multipliers of different operand bit-width n ranging from 4 up to 64 bits. The instances include both signed and unsigned multiplication. The synthesized multipliers are highly optimized and contain a couple of custom-designed components for computing the Booth-encoded partial products. Moreover, due to the use of some *don't care* conditions in the cone of influence of the addition network, also, the logic for some addition steps is optimized at the gate level, and corresponding custom-designed components are instantiated within the network.

Overall, the suite consists of 1040 SMT instances. The quota of solved formulas within specific time-out limits is given in Table 6.2. The quota of instances solved by STABLE and compared against the quotas for other solvers is illustrated with the plots in Figure 6.3. It should be noted that STABLE is the only solver that can handle instances of realistic size. To illustrate this, some representative instances were picked with the detailed results for them as reported in Table 6.3. Here, each name of the instance, also, indicates the operand width and the data type (Booth-encoded signed/unsigned) in the first column. The CPU times of the solvers under comparison are listed in the subsequent columns.

CPU time, sec.	Spear	Boolector	MathSAT	simplifyingSTP	STABLE
< 10	10,8%	13,7%	14,4%	13,4%	72,7%
< 100	19,9%	25,1%	25,9%	21,6%	98,4%
< 250	23,6%	29,3%	29,5%	23,8%	98,4%
< 500	24,3%	32,1%	32,3%	26,5%	99%
< 1000	25,3%	33,2%	36,4%	29,5%	100%
MO/TO/U	74,7%	66,8%	63,6%	70,5%	0%

Table 6.2: Quota for module generator instances

Instance	Spear	Boolector	MathSAT	simplifyingSTP	STABLE
mult_ub_8x8	35	15	20	17	0.6
mult_ub_16x16	TO	TO	TO	TO	4.3
mult_ub_32x32	TO	TO	TO	TO	43
mult_ub_64x64	TO	TO	TO	TO	702
mult_sb_8x8	40	13	30	14	0.4
mult_sb_16x16	TO	TO	TO	TO	2.5
mult_sb_32x32	TO	TO	TO	TO	23
mult_sb_64x64	TO	TO	TO	TO	333

Table 6.3: Selected module generator instances, CPU times, sec.

All instances in this experiment require the extraction technique presented in Section 5.2 to exploit the power of the algebraic proof engine. The high-level rewriting techniques implemented in the competing solvers are obviously not strong enough to provide similar results. Thus, the solvers run into the same performance bottleneck as classical SAT-solvers do on bit-blasted versions of the problems.

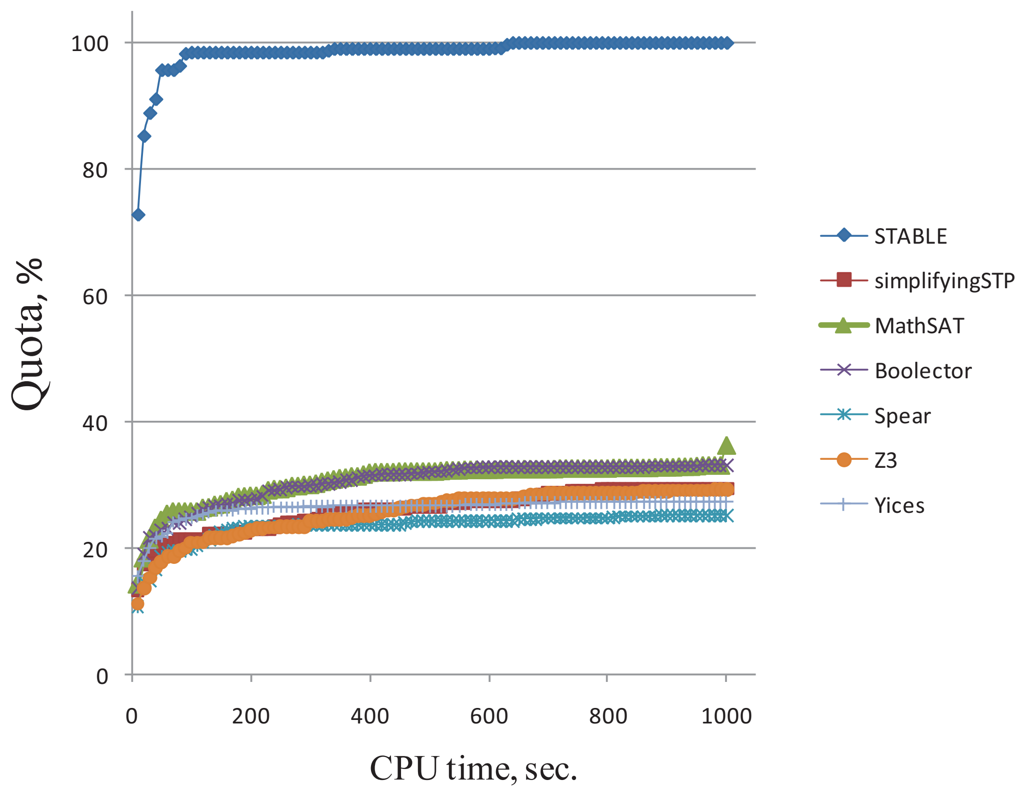


Figure 6.3: Quota of SMT formulas generated from property checking instances of combinatorial multiplier designs and solved by STABLE and its competitors / TO - 1000 sec.

6.2.3 TriCore Benchmark

The second suite of industrial SMT-instances used in the experiments originates from the formal verification of *Infineon's TriCore* micro-processor [Inf]. The instances include the complete data path of the integer pipeline including the sophisticated control logic that allows for the execution of hundreds of instruction variants with multiplication on the same data path. In total, a set of 640 *interval property checking (IPC)* instances was generated. The properties used to verify this data path assume a specific instruction in a specific variant, being decoded in the decode stage of the processor, and prove that the expected arithmetic result is computed in the subsequent clock cycles. The assumption implies dedicated customization of the data path such that the arithmetic components of the data path are connected appropriately to perform the correct computations. For the resulting instances, the propagation step in preprocessing is crucial because it eliminates large portions of the control logic and avoids multiple iterations of the main loop in our solving algorithm.

STABLE solved the complete suite out of 640 SMT-formulas for the TriCore properties successfully in about five hours. Some selected data-path properties and the run-times of STABLE on proving these properties are collected in Table 6.4. Tables B.1 - B.6 list complete experimental data for STABLE. In contrast to that, except for Boolector, the other solvers failed on all of the instances. A closer look at the particular instances reveals that Boolector was able to solve those instances which form instructions where one factor is constant. In this case, the multiplier degenerates to a cascade of adders that Boolector can handle. However, for proper multiplication, the competitors of STABLE give up.

The quota of instances solved by each SMT solver is shown in Table 6.5. The plots comparing quotas of the examined SMT solvers is depicted in Figure 6.4.

Data-path property	CPU time, sec.
res[31:0] = op3[31:0]+(op1[31:0]*op2[31:0])	24
res[63:32] = op3[63:32]+((op1[31:16]*op2[15:0]) $\ll$ 1)[31:0] res[31:0] = op3[31:0]+((op1[15:0]*op2[31:16]) $\ll$ 1)[31:0]	27
res[63:0] = op3[63:0]+ (op1[31:16]*op2[31:16]) $\ll$ 16 + (op1[15:0]*op2[15:0]) $\ll$ 16	11
res[63:32] = op3[63:32]+((op1[31:16]*op2[31:16]) $\ll$ 1)[31:0] res[31:0] = op3[31:0]- (op1[15:0]*op2[15:0])$\ll$ 1)[31:0]	23
res[63:0] = (op1[31:16]*op2[31:16]) $\ll$ 1 + (op1[15:0]*op2[15:0]) $\ll$ 1	11
res[63:0] = op1[31:0]*op2[31:0]	24
res[31:0] = rnd16(((op1*op2[31:16]) $\ll$ 1)[31:0])	12

Table 6.4: Selected results for STABLE over TriCore SMT formulas

CPU time, sec.	Spear	Boolector	MathSAT	simplifyingSTP	STABLE
< 10	0,5%	12%	0,3%	0%	18,4%
< 100	0,5%	12%	0,5%	0%	96,4%
< 250	0,5%	15,8%	0,5%	0%	100%
< 500	0,5%	17,4%	0,5%	0%	100%
< 1000	0,5%	17,7%	0,5%	0%	100%
MO/TO/U	99,5%	82,3%	99,5%	100%	0%

Table 6.5: Quota for TriCore instances

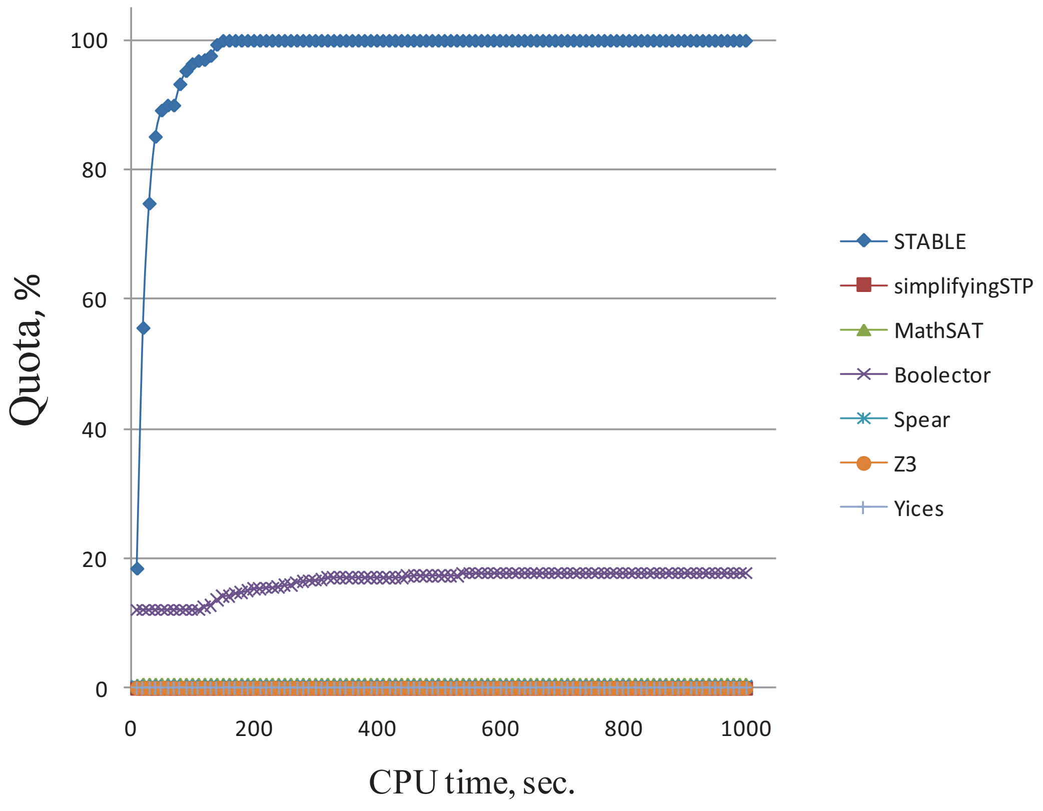


Figure 6.4: Quota of QF-BV SMT formulas generated for data-path verification of processor TriCore and solved by STABLE and its competitors / TO - 1000 sec.

6.2.4 Benchmark of Satisfiable Instances

In this chapter, the evaluation of STABLE concludes with the report on performance for satisfiable instances in formal data-path verification domain. To obtain such instances, typical bugs such as missing or wrong connections, wrong operators, etc. have been introduced into the design of the instances derived from the above mentioned sources.

The graphical representation of the actual quotas distribution is illustrated with the plots in Figure 6.5. The numerical data of quotas for CPU times of different solvers is given in Table 6.6.

It turns out that most of the instances become rather easy to solve such that both STABLE and Boolector can solve all of them. Only Spear and MathSAT fail on a few examples. This is consistent with the industrial experience that SAT quickly finds counterexamples (if they exist) for data-path properties but fails by time-out at proving valid properties.

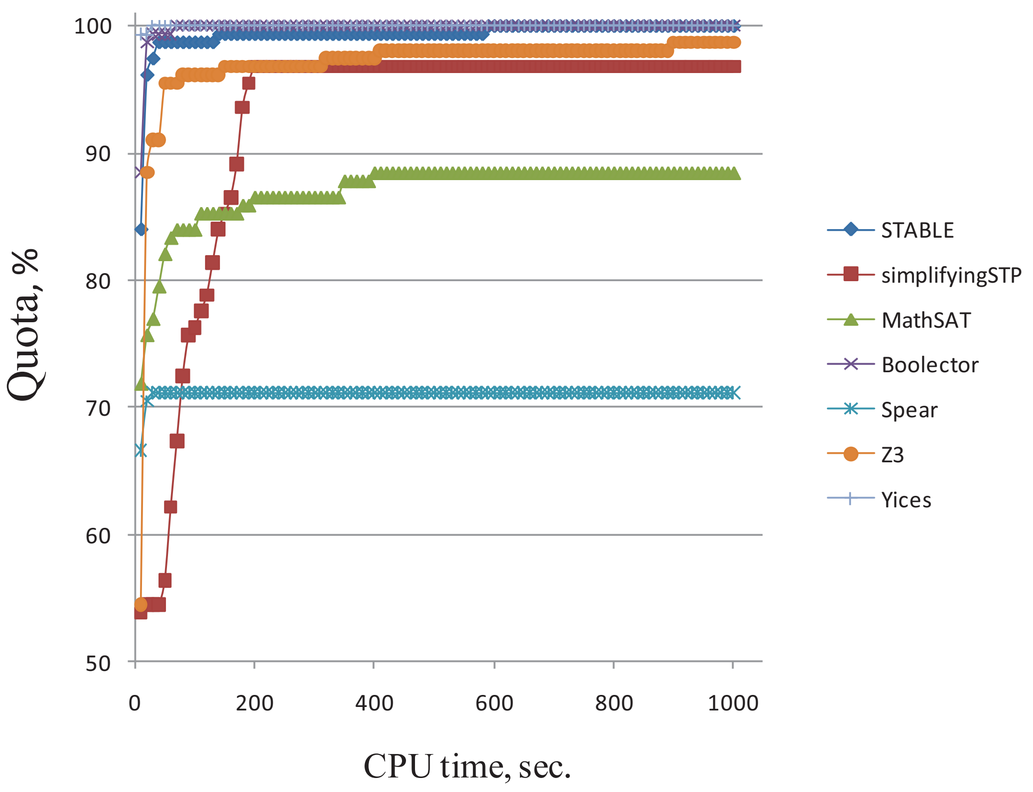


Figure 6.5: Quota of satisfiable QF-BV SMT formulas solved by STABLE and its competitors / TO - 1000 sec.

CPU time, sec.	Spear	Boolector	MathSAT	simplifyingSTP	STABLE
< 10	66,7%	88,5%	71,8%	53,9%	84%
< 100	71,2%	100%	84%	76,3%	98,8%
< 250	71,2%	100%	86,5%	96,8%	99,4%
< 500	71,2%	100%	88,5%	96,8%	99,4%
< 1000	71,2%	100%	88,5%	96,8%	100%
MO/TO/U	28,8%	0%	11,5%	3,2%	0%

Table 6.6: Quota for satisfiable instances

Chapter 7

Summary and Future Work

7.1 Summary

The increasing complexity of modern SoC designs makes tasks of SoC formal verification a lot more complex and challenging. This motivates the research community to develop more robust approaches that enable efficient formal verification for such designs.

It is a common scenario to apply a *correctness by integration* strategy while a SoC design is being verified. This strategy assumes formal verification to be implemented in two major steps. First of all, each module of a SoC is considered and verified separately from the other blocks of the system. At the second step – when the functional correctness is successfully proved for every individual module – the communicational behavior has to be verified between all the modules of the SoC.

In industrial applications, SAT/SMT-based *interval property checking(IPC)* has become widely adopted for SoC verification. Using IPC approaches, a verification engineer is able to afford solving a wide range of important verification problems and proving functional correctness of diverse complex components in a modern SoC design. However, there exist critical parts of a design where formal methods often lack their robustness. State-of-the-art property checkers fail in proving correctness for a data path of an industrial *central processing unit (CPU)*.

In particular, arithmetic circuits of a realistic size (32 bits or 64 bits) – especially implementing multiplication algorithms – are well-known examples when SAT/SMT-based formal verification may reach its capacity very fast. In cases like this, formal verification is replaced with simulation-based approaches in practice. Simulation is a good methodology that may assure a high rate of discovered bugs hidden in a SoC design. However, in contrast to formal methods, a simulation-based technique cannot guarantee the absence of errors in a design. Thus, simulation may still miss some so-called corner-case bugs in the design. This may potentially lead to additional and very expensive costs in terms of time, effort, and investments spent for redesigns, refabrications, and reshipments of new chips.

The work of this thesis concentrates on studying and developing robust algorithms for solving hard arithmetic decision problems. Such decision problems often originate from a task of RTL property checking for data-path designs. Proving properties of those designs can efficiently be performed by solving SMT decision problems formulated with the *quantifier-free logic over fixed-sized bit vectors (QF-BV)*. The highlights and achievements of the work presented in this thesis are briefly summarized in Sections 7.1.1 - 7.1.3.

7.1.1 Algebraic Approach for Verification of Arithmetic Designs

High performance data paths are usually designed at a level of abstraction that is known as the *arithmetic bit level (ABL)*. Chapter 4 presents an approach to efficiently solve decision problems at the arithmetic bit level. This approach combines the technique [WSBK07] with computer algebra algorithms of a Gröbner basis theory over finite rings $\mathbb{Z}/2^n$. In particular, Chapter 4 demonstrates how the ABL decision problems of [WSBK07] can be transformed into a set of variety subset problems. Under certain monomial orderings, the set G of polynomials, generated from the ABL components, forms a Gröbner basis of the ideal $I = \langle G \rangle$ generated by these polynomials. This allows to efficiently solve the variety subset problem and decide problems at the arithmetic bit level. In Chapter 4, it is assumed that the arithmetic problem parts are specified at the arithmetic bit level or word level. In case of custom-designed components, i.e., those parts of an arithmetic design that are specified below the ABL, a sophisticated technique is required to restore all the necessary arithmetic description from these components. Such extraction techniques are presented in Chapter 5.

In contrast to other algebraic techniques applied for data-path verification in the past, the approach of Chapter 4 neither needs to implement a time-consuming generation of a Gröbner basis nor performs an expensive zero-function test.

7.1.2 ABL Modeling of Custom-Designed Components

The ABL normalization [WSBK07] and the technique of computer algebra [PWS$^+$11] have been proven viable approaches to formal property checking of data-path designs. They are applicable where arithmetic components and subcomponents can be identified on the *register-transfer level (RTL)* of the design and the property. Chapter 5 extends the applicability of these approaches to cases where some of the arithmetic components are custom-designed entities, e.g., specified using Boolean equations or gates. Two techniques are presented for transformation of these entities into arithmetic building blocks.

The first technique is developed to increase capacity of the ABL normalization. This technique is based on the Reed-Muller expansion and uses Reed-Muller expressions as intermediate representations for Boolean functions. Section 5.2 describes how Boolean logic, expressed in such Reed-Muller forms, can automatically be transformed into ABL components so that such logic blocks can further be treated together with the remaining ABL components in a subsequent normalization run. As demonstrated with experimental results in Section 5.4, this extraction technique was successfully evaluated on a number of industrial designs generated by a commercial arithmetic module generator.

The second technique proposed in Chapter 5 is a modification of the extraction algorithm based on the Reed-Muller expansion. The modified version of the technique enables an efficient creation of functionally equivalent arithmetic polynomials over Boolean variables for a gate netlist. This technique may greatly enhance the applicability of the algebraic approach. As a result, it allows to build a stronger framework to successfully solve hard arithmetic decision problems in property checking for SoCs of a custom-designed flow. In practice, this extraction technique was conjoined with the approach of computer algebra and both were integrated into a common SMT solver as described in Chapter 6.

7.1.3 New QF-BV SMT Solver

The main application domain for the techniques discussed in this thesis is formal verification of SoC modules designed for complex computational tasks, for example, in signal processing applications. Ensuring proper functional behavior for such modules, including arithmetic correctness of the data paths, is considered a very difficult problem for standard SAT/SMT-based property checking.

Chapter 6 explains how methods from computer algebra can be integrated into an SMT solver such that instances can be handled where the arithmetic problem parts are specified mixing various levels of abstraction from the plain gate level for small highly optimized components up to the pure word level used in high-level specifications. In Chapter 6, a new SMT solver STABLE is introduced for the quantifier-free logic over fixed-sized bit vectors. This solver integrates the proposed algebraic technique of Chapter 4 and the extraction technique of Section 5.3 to solve hard arithmetic decision problems. In STABLE, a standard SAT solver is used as a back end for solving the non-arithmetic problem parts.

The experiments demonstrate high effectiveness and robustness of STABLE in proving functional correctness for arithmetic designs of a realistic size in comparison with other contemporary QF-BV SMT solvers.

7.2 Future Work

This thesis concentrated on efficient solving of hard arithmetic decision problems that are frequently encountered in formal verification of RTL data-path designs. In particular, the thesis proposed new methodology to effectively tackle such verification problems in practice. The next sections describe extensions for the proposed methodology and outline research for the future work.

7.2.1 Smart and lazy ABL Extraction

As follows from Chapter 5, the ABL extraction algorithms are crucial for an appropriate modeling of arithmetic decision problems in cases when an RTL design, heavily optimized at the logic level, is to be formally verified by means of the algebraic approach from Chapter 4. Moreover, the number of polynomials and their compactness have a direct impact on the efficiency of the algebraic approach to calculate a reduced normal form of the proof goal. To be more precise, the more compact a system of polynomial equations is provided, the faster the approach processes it and the less memory is, thereby, consumed.

The approaches of Chapter 5 use greedy ABL extractions. For a given gate netlist, at first, they generate arithmetic polynomials for each gate. As the next step, the polynomials defining the primary outputs of the netlist have to be normalized up to the primary inputs. The pre-normalized polynomials are, furthermore, analyzed to compute (if possible) compact ABL models thereof.

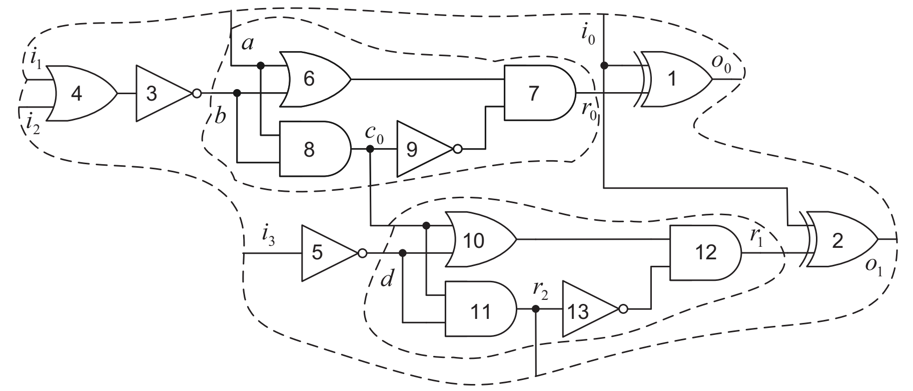


Figure 7.1: Example of a gate netlist for an arithmetic custom-designed component

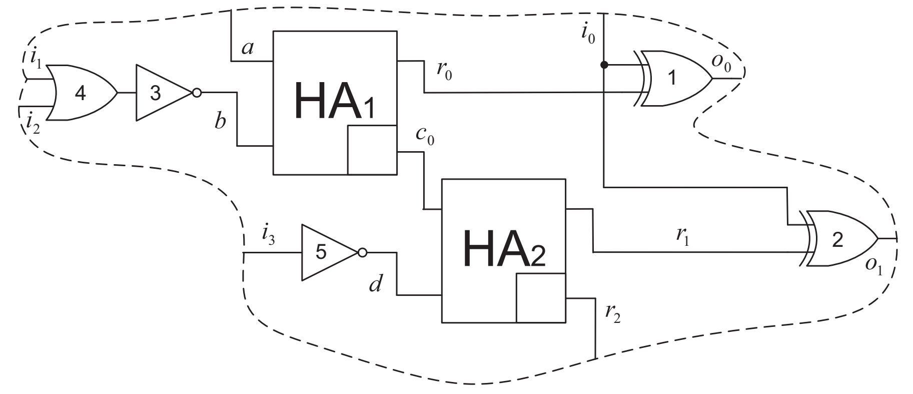
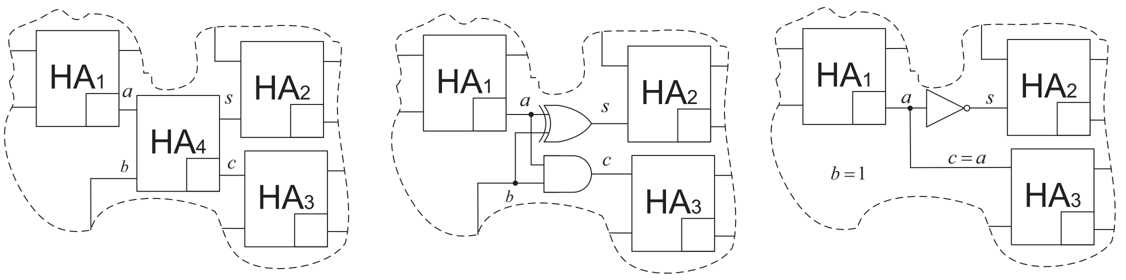


Figure 7.2: Result of a lazy ABL extraction for the example of Figure 7.1

The compactness of ABL models may significantly be increased if all arithmetic functional units are successfully identified throughout the gate netlist under investigation. This may not always be achieved with a greedy ABL extraction when a gate netlist consists of logic parts with mixed arithmetic functions, where the bounds between functional units are blurred at the logic level. Figure 7.1 demonstrates this problem. Two half adders, implemented by the gates $6, \ldots, 13$, are surrounded with other logic elements designated with $1, \ldots, 5$. A greedy extraction will miss the adders and simply end with three long polynomial functions $o_0 = f_0(a, i_0, i_1, i_2)$, $o_1 = f_1(a, i_0, i_1, i_2, i_3)$, and $r_2 = f_2(a, i_1, i_2, i_3)$. Further reduction of these polynomials becomes impossible. Therefore, in cases like this, an advanced extraction algorithm is needed. It should be a kind of a lazy approach that considers, at the beginning, pieces of logic circuitry to find exact arithmetic components, e.g., half adders $HA_1 : r_0 + 2c_0 = a + b$ and $HA_2 : r_1 + 2r_2 = c_0 + d$ in Figure 7.2. Only after that a greedy extraction may be applied. Note, the example of Figure 7.2 demonstrates one more potential improvement followed by a lazy extraction. The successfully identified half adders may later be combined into an addition network of one polynomial $r_0 + 2r_1 + 4r_2 = a + b + 2d$. In practice, such reductions might greatly simplify the overall polynomial system.



(a) (b) (c)

Figure 7.3: Gate-level optimization of the half adder *HA*$_4$

The other important issue that an advanced extraction approach must take into account is the identification of arithmetic entities in a gate netlist when logic optimizations due to don't care conditions, internal constant values, internal equivalences, etc. have been applied. Sometimes, such optimizations may substantially restructure a gate netlist so that an extraction of individual functional components becomes a very complex task. For instance, Figure 7.3 depicts refinement steps of the logic circuit for the half adder $HA_4 : s + 2c = a + b$ when one of its input operands has the constant value. The reverse development of this circuit into its ABL model has to be treated in a special manner. For example, the merging of the reverse engineering approach described in Section 3.2.1 and the extraction algorithms of Chapter 5 might be a possible solution to fix this problem.

7.2.2 ABL Proof Logging

In designing and implementing of an SMT solver, a lion's share of all costs and efforts falls to testing and debugging of the tool. While tests can often be executed automatically, the debugging is usually a manual, complex and even error-prone task.

Therefore, a certified proof is desired to validate the correctness of algorithmic implementations that are responsible to perform computations inside a solver. For instance, in an ABL-based SMT solver, the implementation of an underlying ABL engine was not

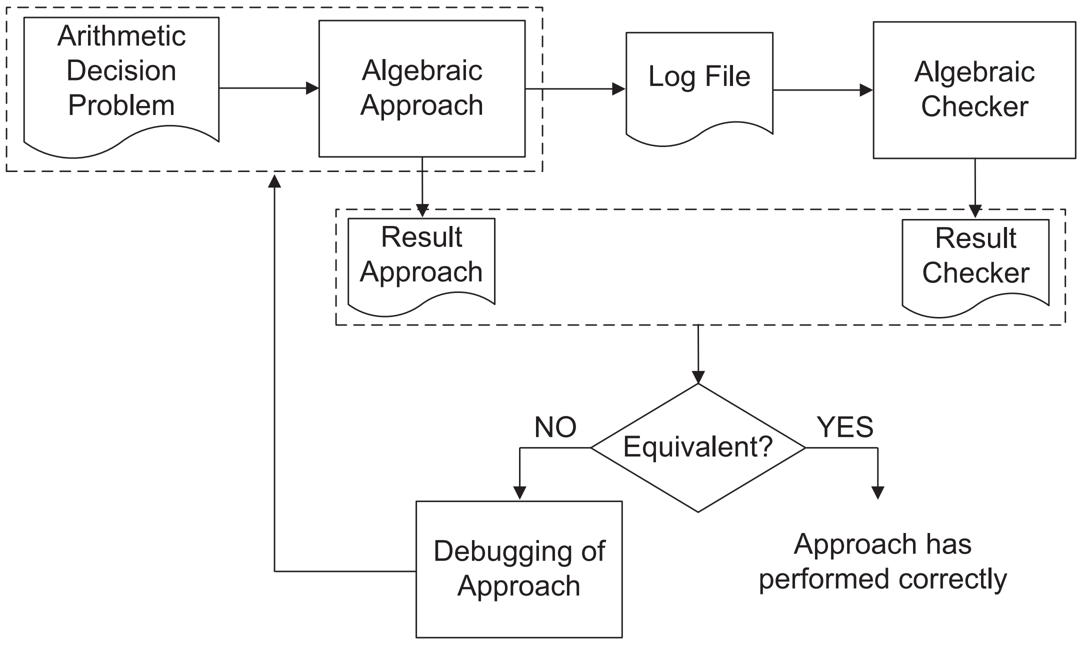


Figure 7.4: Flow of ABL proof logging

yet appropriately certified, although, in the last decades, different research groups have proposed a variety of ABL approaches that experimentally proved to be robust enough to tackle hard arithmetic decision problems. Indeed, the way for proper semi-/automatic testing and debugging of such approaches was so far not discussed at all. So, the development of techniques that enable efficient testing and debugging in the ABL domain is still a challenge.

A possible solution might be a technique for ABL proof logging as shown in Figure 7.4. The idea is the same as with proof traces that may be used for debugging of SAT solvers. For example, such traces can be generated with the BDD-based SAT solver *EBDDRES* [EBD] that for an input CNF performs an extended resolution proof as described in [SB06, JSB06]. Similar to EBDDRES, the algebraic approach under test has to be able to dump a log file with traces of all algebraic calculation steps being made while an input problem is solved. In the sequel, the data of this file has to be analyzed with an alternative algebraic checker. For instance, SINGULAR [GPS10] is a suitable candidate to be a good checker for the QF-BV solver STABLE. If the result produced by such a checker does not coincide with the result computed by the approach under test then this implementation of the approach is erroneous and must be debugged correspondingly.

It is apparent that a fully automatic ABL proof logging is not feasible in practice, since debugging always needs manual interventions of a designer. But still, the rest of the ABL-proof-logging flow from Figure 7.4 can easily be automated. For this purpose, it is important to develop a common proof-trace format and to create a well-defined interface among all the engines that have to be integrated into the proof-logging environment.

Chapter 8

Zusammenfassung

Mit zunehmendem Umfang von modernen SoC-Entwürfen (engl.: *System-on-Chip designs*) steigt auch die Komplexität der formalen Verifikation von SoC-Entwürfen. Das motiviert die Forschungsgemeinschaft robustere Verfahren zu entwickeln, die eine effiziente Verifikation von solchen Entwürfen ermöglichen.

In der Praxis wird bei der Verifikation deshalb häufig eine Strategie angewendet, bei der Korrektheit durch Integration (engl.: *correctness by integration*) erreicht wird. Diese Strategie führt die formale Verifikation in zwei Schritten aus. Zuerst konzentriert man sich auf die Verifikation einzelner Module in einem SoC-Entwurf. Sobald die Korrektheit für jedes Modul erfolgreich bewiesen worden ist, beginnt man mit der Verifikation des ganzen Kommunikationsverhaltens zwischen den einzelnen Komponenten im System.

In industriellen Anwendungen ist die SAT/SMT-basierte Intervall-Eigenschaftsprüfung (engl.: *interval property checking (IPC)*) heutzutage zu einer der meistverwendeten Techniken der SoC-Verifikation geworden. Mittels IPC kann man ein breites Spektrum von wichtigen Verifikationsproblemen lösen und darüber hinaus diverse komplexe Einheiten auf ihre funktionelle Korrektheit in einem SoC-Entwurf untersuchen. Es gibt jedoch kritische Teile in einem RTL-Design (engl.: *register-transfer-level design*), wo die Robustheit für formale Verifikationsverfahren noch immer fehlt. So scheitern zum Beispiel moderne Property Checker an formalen Korrektheitsbeweisen für Datenpfade eines industriellen Prozessors (engl.: *central processing unit (CPU)*).

Insbesondere arithmetische Schaltungen einer realistischen Größe, beispielweise solche arithmetische Einheiten wie ein Multiplizierer mit Datenwortbreiten von 32 Bits oder 64 Bits, sind wohlbekannte Beispiele, wo die formale Verifikation ihre Leistungsgrenze sehr schnell erreicht. In Fällen wie diesen wird die formale Verifikation in der Regel durch simulationsbasierte Techniken ersetzt. Simulation ist ein praktikabler Ansatz für die Analyse und das Debugging von RTL-Entwürfen. In der Praxis unter Verwendung von Simulation ist die Anzahl von entdeckten Entwurfsfehlern meistens sehr hoch. Allerdings, im Gegensatz zu formaler Verifikation, sind simulationsbasierte Methoden unfähig, die Fehlerfreiheit für große RTL-Designs zu garantieren. Einige Fehler können dabei übersehen werden. Die Kosten, die von zu spät entdeckten Fehlern verursacht werden, können immens sein.

Der Schwerpunkt dieser Arbeit richtet sich auf die Untersuchung und auf die Entwicklung robuster Algorithmen, um schwere arithmetische Entscheidungsprobleme effektiv zu lösen. Solche Probleme entstehen oft bei einer RTL-Eigenschaftsprüfung für industrielle Datenpfadentwürfe. Eigenschaftsprüfung beruht in der Praxis meistens auf SAT/SMT-basierten Methodiken. Für die Formulierung derartiger Verifikationsprobleme eignet sich die nicht quantifizierte Logik mit Bitvektoren fixer Bitbreite (engl.: *quantifier-free logic over fixed-sized bit vectors (QF-BV)*) gut. Diese Logik wird unter anderem durch das SMT-LIB-Format [RT06, BST10] standardisiert.

In letzter Zeit haben verschiedene Arbeitsgruppen spezielle Tools entwickelt, die sich mit dem Lösen von QF-BV-Entscheidungsproblemen befassen. Es handelt sich dabei um sogenannte SMT-Solver wie beispielweise *Spear* [BH08, Spe], *Boolector* [BB09, boo], *MathSAT* [BBC+05], *simplifyingSTP* bezogen auf die Version 939 von *STP* [GD07,STP], *Z3* [Z3,MB08] und *Yices* [Yic,DM06,DdM06]. Wie die Ergebnisse der SMT-Wettbewerbe [SMT07, SMT08, SMT09, SMT10] zeigen, sind diese Solver im Allgemeinen sehr leistungsfähig mit der Ausnahme der Behandlung schwerer arithmetischer Probleme. Diese stellen bedauerlicherweise einen Flaschenhals für die vorgeschlagenen SMT-Solver dar.

Der Hauptbeitrag dieser Arbeit ist die Entwicklung eines effizienten Verfahrens, welches das erfolgreiche Behandeln von schweren arithmetischen Entscheidungsproblemen ermöglicht. Dazu wurden zwei wichtige Ansätze vorgeschlagen. Der erste Ansatz beschreibt ein algebraisches Verfahren, das das Folgende erzielt:

- die Modellierung eines arithmetischen Entscheidungsproblems durch eine Menge von algebraischen Polynomen im Ring $Q := \mathbb{Z}/2^N[X]/\langle x^2 - x \ : \ x \in X\rangle$,
- die weitere Lösung des Problems durch die Normalformberechnung im Ring Q.

Der zweite Ansatz schlägt einen Extraktionsalgorithmus vor und zeigt, wie man die logische Beschreibung eines arithmetischen Teilproblems in eine Menge arithmetischer Polynome umwandeln kann. Diese Teilprobleme tauchen oftmals in der Praxis auf, hauptsächlich wenn ein zu verifizierendes RTL-Design einige Komponenten enthielt, die auf Boolescher Ebene einer Tiefenoptimierung unterzogen wurden. Beide Ansätze wurden in einen neuen QF-BV-SMT-Solver integriert, genannt *STABLE.*

8.1 Algebraisches Verfahren

Leistungsfähige RTL-Datenpfade sind üblicherweise auf einer Abstraktionsebene entworfen, die als arithmetische Bitebene (engl.: *arithmetic bit level (ABL)*) bezeichnet wird. Die dabei entstehenden Verifikationsprobleme kann man auch auf dieser Ebene kompakt darstellen. Im Folgenden entwickeln wir eine Verifikationstechnik, die solche ABL-Informationen ausnutzt. Damit erhalten wir ein formales Verfahren, das eine effektive und robuste Verifikation erfolgreich gewährleistet.

Kapitel 4 stellt ein Verfahren für die effektive Behandlung von Entscheidungsproblemen auf der arithmetischen Ebene vor. Das Verfahren kombiniert die Technik [WSBK07] mit den computeralgebraischen Algorithmen, die auf Gröbnerbasentheorien über endlichen Ringen $\mathbb{Z}/2^N$ basieren. Kapitel 4 zeigt, wie man Probleme der Eigenschaftsprüfung auf der ABL durch algebraische Polynome modelliert. Dabei wird jede ABL-Komponente g_i in eine Menge G_i von arithmetischen Polynomen im Polynomring $\mathbb{Z}/2^N[x_1, \ldots, x_n]$ übersetzt. Unter einer bestimmten Monomordnung bildet die Menge G aller generierten Polynome eine starke Gröbnerbasis für ein Ideal $I = \langle G \rangle$. Sei mit g das Polynom der Zielsetzung bezeichnet. Dann kann man, wie in Kapitel 4 erläutert ist, die ABL-Entscheidungsprobleme aus [WSBK07] durch eine Menge von Varietätsteilmengenproblemen (engl.: *variety subset problems*) darstellen. Das heißt, man muss zeigen, dass

$$V(\langle G \rangle) \subseteq V(g) \tag{8.1}$$

gilt, wobei V für die Varietät steht. Im Allgemein ist die Varietät eines Polynoms die Menge aller Nullstellen dieses Polynoms. Auf diese Weise ist es offensichtlich, dass das vorgestellte algebraische Verfahren sämtliche Wertebelegungen für alle Variablen aus der Menge G erfasst. Die Bedingung 8.1 ist dann und nur dann erfüllt, wenn die nachstehende Gleichung gilt:

$$NF(2^{N-n}g, G) = 0. \tag{8.2}$$

Das heißt, es muss sich ein Nullpolynom für die reduzierte Normalform von g im Bezug auf G ergeben. Die Algorithmen der Computeralgebra erlauben eine effektive Normalformberechnung und gestatten damit, Probleme auf der arithmetischen Abstraktionsebene effektiv zu entscheiden.

Im Unterschied zu anderen algebraischen Techniken, die man für die Verifikation von Datenpfaden benutzt, hat das Verfahren aus Kapitel 4 zwei entscheidende Vorteile:

- keine zeitaufwendige Generierung von Gröbnerbasen,
- kein teurer Nullfunktiontest.

Das Beispiel aus Abschnitt 4.3 demonstriert den Ablauf des Verfahrens durch den vorgelegten Beweis auf die Korrektheit für den Entwurf eines kombinatorischen Multiplizierers.

In Kapitel 4 wird davon ausgegangen, dass die Teile des arithmetischen Problems auf der arithmetischen Ebene und/oder auf der Wortebene angegeben sind. Anwendungsspezifische Komponenten (engl.: *custom-designed components*) werden in der Regel auf der logischen Gatterebene, also unterhalb der ABL-Abstraktionsebene, entworfen. Dadurch können noch bessere Optimierungsergebnisse erzielt werden. Man benötigt dann jedoch eine elegante Technik, um die erforderliche ABL-Beschreibung für eine solche Gatternetzliste wieder herzustellen. Eine solche Extraktionstechnik ist in Kapitel 5 detailliert beschrieben. Wie in Kapitel 6 berichtet ist, kann das neue algebraischen Verfahren zusammen mit der Extraktionstechnik erfolgreich kombiniert werden.

8.2 Algebraische Modellierung für logische Constraints

Mit der ABL-Normalisierung [WSBK07] und deren Weiterentwicklungen mit Techniken der Computeralgebra [WWS$^+$08, PWS$^+$11] liegen effektive Verfahren für die formale Eigenschaftsprüfung von Datenpfaden vor. Diese Verfahren sind gut anwendbar, wenn die arithmetischen Komponenten eines Entwurfs auf der arithmetischen Bitebene oder auf der Wortebene spezifiziert sind. Kapitel 5 erweitert die Verwendungsmöglichkeit dieser Verfahren auf Fälle, in denen einige arithmetische Komponenten auf der logischen Gatterebene optimiert worden sind, d.h. diese Komponenten sind durch Boolesche Gleichungen oder Gatter spezifiziert. Beispiele für solche Komponenten könnten ein Booth-Encoder oder verfeinerte Additionsanordnungen sein, die üblicherweise unterhalb der ABL-Abstraktionsebene implementiert werden. In realen industriellen Probleminstanzen bilden solche anwendungsspezifischen Komponenten zusammen mit ABL-Komponenten eine monolithische arithmetische Funktionseinheit. Daher enthalten in der Praxis manche Bestandteile des Entwurfs handgefertigte Optimierung und beinhalten spezialisierte Logik, so dass es unmöglich ist, eine RTL-Beschreibung in eine ABL-Beschreibung sofort zu übersetzen. Kapitel 5 zeigt eine Methode, um anwendungsspezifische Komponenten einer arithmetischen Schaltung in eine funktionell äquivalente ABL-Beschreibung zu transformieren. Anschließend lässt sich die Anwendung von den ABL-basierten Algorithmen, die in Kapitel 3, 4 beschrieben worden sind, erfolgreich durchführen. Somit füllt der vorgeschlagene Ansatz eine wesentliche Lücke im Ablauf der formalen Verifikation und stellt damit sicher, dass das manuelle Verfahren [KJW$^+$08] aus Abschnitt 3.2.1 für High-End-Anwendungsfälle verwendet werden kann. Kapitel 5 präsentiert zwei Techniken, mit denen man anwendungsspezifische Komponenten in arithmetische Bausteine umwandeln kann.

Der erste Ansatz wurde entwickelt, um die Leistungsfähigkeit der ABL-basierten Normalisierung zu verbessern. Er beruht auf der Reed-Muller-Zerlegung und ist in Abschnitt 5.2 dargestellt. Hier werden Reed-Muller-Formen als eine Zwischendarstellung für Boolesche Funktionen verwendet. Abschnitt 5.2 zeigt, wie Boolesche Logik, die durch Reed-Muller-Formen ausgedrückt ist, automatisch in ABL-Komponenten umgewandelt werden kann, so dass solche logischen Blöcke zusammen mit den übrigen ABL-Komponenten im folgenden Normalisierungsablauf behandelt werden können. Die experimentellen Ergebnisse aus Abschnitt 5.4 zeigen, dass das Extraktionsverfahren für etliche Designs in effektiver Weise angewendet werden kann. Diese Designs wurden mit einem kommerziellen Generator für arithmetische Module erstellt.

Der zweite vorgeschlagene Ansatz aus Kapitel 5 ist eine Modifikation des Extraktionsalgorithmus, basierend auf einer Reed-Muller-Zerlegung. Für eine gegebene Gatternetzliste ermöglicht diese geänderte Variante des Extraktionsansatzes eine effiziente Erstellung von funktionell äquivalenten arithmetischen Polynomen mit Booleschen Variablen. Abschnitt 5.3 entwickelt das Konzept der ABL-Synthese aus Abschnitt 5.2 weiter und stellt ein Verfahren dar, um normalisierte Polynome in einem Ring $\mathbb{Z}/2^N$ zu berechnen, damit die Algorithmen der Computeralgebra aus Kapitel 4 angewendet werden können. Um die Bearbeitung von nicht-arithmetischen Constraints mit dem algebraischen Verfahren zu ermöglichen, wurde eine Technik erarbeitet, die arithmetische Polynome generiert, welche die arithmetische Funktion nicht-arithmetischer Constraints modellieren. Für eine Netzliste von Booleschen Constraints lassen sich solche Polynome mit dieser Technik in zwei aufeinanderfolgenden Phasen gewinnen:

- zuerst werden Reed-Muller-Formen und arithmetische Umformungen verwendet, um polynomische Gleichungen für jeden Booleschen Constraint abzuleiten,
- dann werden alle Polynome bezüglich der Eingangsvariablen der Netzliste normalisiert.

Diese Technik erlaubt das algebraische Verfahren auch dann einzusetzen, wenn keine vollständige arithmetische Beschreibung des Entwurfs existiert. Die experimentellen Daten für diese Technik sind in Kapitel 6 dargestellt. Hier ist auch erläutert, wie man diese Technik mit dem algebraischen Verfahren aus Kapitel 4 integrieren kann, um SMT-Entscheidungsprobleme zu lösen, die in der Praxis bei der formalen Verifikation von arithmetischen Designs entstehen.

8.3 QF-BV-SMT-Solver

Das Hauptanwendungsgebiet der vorgestellten Techniken ist die formale Verifikation von SoC-Modulen, die man für die Aufgaben einer hohen Berechnungskomplexität entworfen hat, wie z.B. bei der Signalverarbeitung. Das korrekte funktionelle Verhalten solcher Module zu gewährleisten und insbesondere die arithmetische Korrektheit für Datenpfade zu beweisen, ist ein schweres Problem für einen Standard-SAT/SMT-basierten Eigenschaftsprüfer.

Kapitel 6 beschreibt den neuen SMT-Solver STABLE. Dieser Solver kombiniert die beiden entwickelten Techniken, und zwar das in Kapitel 4 eingeführte computeralgebrabasierte Verfahren und die in Kapitel 5 diskutierte Extraktionsmethode. Im Hintergrund von STABLE wird ein normaler SAT-Solver eingesetzt, um das Lösen nicht-arithmetischer Teilprobleme zu erlauben. STABLE wurde zunächst in [WPD$^+$10, PWS$^+$11] vorgestellt. Er ist vor allem aufgebaut, um Formeln der nicht quantifizierten Logik mit Bitvektoren fixer Bitbreite (QF-BV) effizient zu behandeln.

Solche Formeln werden häufig bei der Eigenschaftsprüfung für Module von SoC-Entwürfen erzeugt. Bei der Verifikation von Hochleistungsdatenpfadmodulen tauchen oft anwendungsspezifische arithmetische Komponenten auf, die auf der logischen Ebene der jeweiligen Hardwarebeschreibungssprache spezifiziert worden sind. Dies führt zu Verifikationsproblemen, wo einige arithmetische Teile des Beweisproblems nicht-arithmetische Constraints enthalten können. Da die vorgeschlagene Verifikationstechnik auch den Algorithmus beinhaltet, der eine arithmetische Bitebeneinformation für diese Boolesche Constraints extrahieren kann, ist das eingebaute algebraische Verfahren gut anwendbar, um arithmetische Teilprobleme vollständig zu lösen. Die nicht-arithmetischen Teilprobleme sind letztendlich zusammen mit den Ergebnissen des Verfahrens der Computeralgebra in

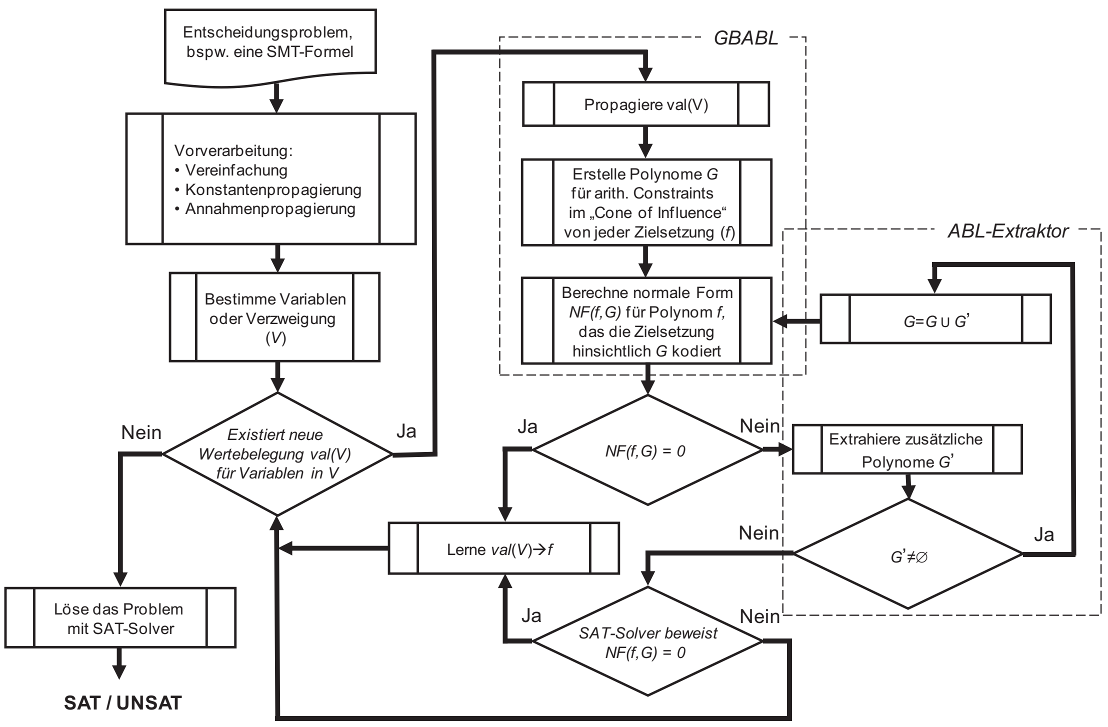

Entscheidungsproblem, bspw. eine SMT-Formel
Vorverarbeitung:
• Vereinfachung
• Konstantenpropagierung
• Annahmenpropagierung
Bestimme Variablen oder Verzweigung (V)
Existiert neue Wertebelegung val(V) für Variablen in V
Nein
Ja
Löse das Problem mit SAT-Solver
SAT / UNSAT
GBABL
Propagiere val(V)
Erstelle Polynome G für arith. Constraints im „Cone of Influence“ von jeder Zielsetzung (f)
Berechne normale Form NF(f,G) für Polynom f, das die Zielsetzung hinsichtlich G kodiert
ABL-Extraktor
G=G ∪ G'
NF(f,G) = 0
Ja
Nein
Extrahiere zusätzliche Polynome G'
G'≠∅
Nein
Ja
Lerne val(V)→f
SAT-Solver beweist NF(f,G) = 0
Ja
Nein

Abbildung 8.1: Das Ablaufdiagramm vom SMT-Solver STABLE für Lösen von QF-BV-Entscheidungsproblemen mit Hilfe vom *GBABL (Gröbner-Basis-basierte-ABL)*-Ansatz und dem *ABL-Extraktor*

eine CNF übersetzt, die weiter mit einem SAT-Solver bearbeitet wird. Der grundlegende Ablaufplan von STABLE ist in Abbildung 8.1 dargestellt.

Wie die Experimente in Kapitel 6 zeigen, ist STABLE höchst leistungsfähig und robust im Vergleich zu anderen modernsten QF-BV-SMT-Solvern wie *Spear-2-7* [BH08, Spe], *Boolector 1.4* [BB09,boo], *MathSAT v. 4.3-smtcomp* [BBC$^+$05], und *simplifyingSTP* basierend auf der Revision 939 von *STP* [GD07, STP]. Als einziger uns bekannter Solver kann STABLE funktionelle Korrektheit für arithmetische Entwürfe einer realistischen Größe wirksam beweisen.

Appendix A

Examples of Source Codes

```
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)
(set-info :status unknown)
(declare-fun var2 () (_ BitVec 2))
(declare-fun bvlambda_1_ () (_ BitVec 1))
(declare-fun bvlambda_2_ () (_ BitVec 1))
(assert (let ((?let_k_0 (concat (_ bv0 2) var2) ))
(let ((?let_k_1 (bvmul ?let_k_0 ((_ extract 3 0) (bvadd (concat (_ bv0
     1) ((_ extract 3 0) (bvadd (_ bv1 5) (bvnot (concat (concat (_ bv0 3)
     bvlambda_1_) (_ bv0 1)))))) (concat (_ bv0 4) bvlambda_2_))))))
(let ((?let_k_2 (bvmul var2 (concat (_ bv0 1) bvlambda_1_))))
(let ((?let_k_3 (bvadd (concat (_ bv0 2) ((_ extract 0 0) ?let_k_2))
     (concat (_ bv0 2) ((_ extract 2 2) ?let_k_1))))) ((and (not false)
     (not (= (_ bv0 1) (bvnot (ite (= (bvmul ?let_k_0 (concat (concat (_
     bv0 2) bvlambda_1_) bvlambda_2_)) (concat (concat ((_ extract 0 0)
     (bvadd (concat (_ bv0 1) ((_ extract 1 0) (bvadd (concat (_ bv0 2) ((_
     extract 1 1) ?let_k_3)) (concat (_ bv0 2) ((_ extract 3 3)
     ?let_k_1))))) (concat (_ bv0 2) ((_ extract 1 1) ?let_k_2)))) ((_
     extract 0 0) ?let_k_3)) ((_ extract 1 0) ?let_k_1))) (_ bv1 1) (_ bv0
     1)))))))))) ) )
)
(check-sat)
(exit)
```

Figure A.1: SMT formula defined in accordance with SMT-LIB version 2.0 [BST10] for property checking instance of (2×2) unsigned integer multiplier represented in Figure A.5

```
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)
(set-info :status unknown)
(declare-fun bvlambda_1_ () (_ BitVec 1))
(declare-fun bvlambda_2_ () (_ BitVec 1))
(declare-fun bvlambda_3_ () (_ BitVec 1))
(declare-fun bvlambda_4_ () (_ BitVec 1))
(assert (let ((?let_k_0 (concat (_ bv0 2) bvlambda_1_) ))
(let ((?let_k_1 (bvnot bvlambda_1_)))
(let ((?let_k_2 (bvand bvlambda_2_ bvlambda_4_)))
(let ((?let_k_3 (bvand bvlambda_1_ (bvnot bvlambda_2_))))
(let ((?let_k_4 (bvand bvlambda_3_ ?let_k_3)))
(let ((?let_k_5 (bvadd (concat (_ bv0 1) ((_ extract 1 0) (bvadd
     (concat (_ bv0 2) ((_ extract 1 1) (bvadd (concat (_ bv0 2) ((_
     extract 1 1) (bvadd ?let_k_0 (concat (_ bv0 2) (bvand (bvnot (bvand
     bvlambda_1_ ?let_k_2)) (bvnot (bvand ?let_k_1 (bvnot ?let_k_2))))))))
     (concat (_ bv0 2) (ite (= bvlambda_1_ (bvand (bvnot (bvand bvlambda_2_
     bvlambda_3_)) (bvnot (bvand bvlambda_4_ ?let_k_3)))) (_ bv1 1) (_ bv0
     1)))))) (concat (_ bv0 2) (bvand (bvnot (bvand bvlambda_1_ ?let_k_4))
     (bvnot (bvand ?let_k_1 (bvnot ?let_k_4)))))))) (concat (_ bv0 2)
     (bvand bvlambda_1_ bvlambda_4_)))))
(let ((?let_k_6 (bvadd (concat (_ bv0 2) ((_ extract 1 1) (bvadd
     ?let_k_0 (concat (_ bv0 2) (bvand (bvnot (bvand bvlambda_1_ ?let_k_2))
     (bvnot (bvand ?let_k_1 (bvnot ?let_k_2)))))))) (concat (_ bv0 2) (ite
     (= bvlambda_1_ (bvand (bvnot (bvand bvlambda_2_ bvlambda_3_)) (bvnot
     (bvand bvlambda_4_ ?let_k_3)))) (_ bv1 1) (_ bv0 1))))))
(let ((?let_k_7 (bvadd ?let_k_0 (concat (_ bv0 2) (bvand (bvnot (bvand
     bvlambda_1_ ?let_k_2)) (bvnot (bvand ?let_k_1 (bvnot ?let_k_2))))))))
     ((and (not false) (not (= (_ bv0 1) (bvnot (ite (= (bvmul (concat
     (concat (_ bv0 2) bvlambda_3_) bvlambda_4_) (concat ?let_k_0
     bvlambda_2_)) (concat (concat (concat (bvnot ((_ extract 0 0) (bvadd
     (concat (_ bv0 1) ((_ extract 1 0) (bvadd (concat (_ bv0 2) ?let_k_1)
     (concat (_ bv0 2) ((_ extract 1 1) ?let_k_5))))) (concat (_ bv0 2)
     (bvand bvlambda_1_ bvlambda_3_))))) ((_ extract 0 0) ?let_k_5)) ((_
     extract 0 0) ?let_k_6)) ((_ extract 0 0) ?let_k_7))) (_ bv1 1) (_ bv0
     1))))))))))))) ) )
)
(check-sat)
(exit)
```

Figure A.2: SMT formula defined in accordance with SMT-LIB version 2.0 [BST10] for property checking instance of (2×2) unsigned integer multiplier represented in Figure A.6

```
(benchmark mult_ub_2x2_abl
:logic QF_BV
:extrafuns ((var2 BitVec[2]))
:extrafuns ((bvlambda_1_ BitVec[1]))
:extrafuns ((bvlambda_2_ BitVec[1]))
:formula
(let (?e2 bv0[2])
(let (?e3 bv0[3])
(let (?e4 bv30[5])
(let (?e5 bv0[4])
(let (?e6 bv0[1])
(let (?e7 (concat ?e2 var2))
(let (?e10 (concat ?e2 bvlambda_1_))
(let (?e11 (concat ?e10 bvlambda_2_))
(let (?e12 (bvmul ?e7 ?e11))
(let (?e13 (concat ?e6 bvlambda_1_))
(let (?e14 (bvmul var2 ?e13))
(let (?e15 (extract[0:0] ?e14))
(let (?e16 (concat ?e2 ?e15))
(let (?e17 (concat ?e3 bvlambda_1_))
(let (?e18 (concat ?e17 ?e6))
(let (?e19 (bvadd (bvnot ?e4) (bvnot ?e18)))
(let (?e20 (extract[3:0] ?e19))
(let (?e21 (concat ?e6 ?e20))
(let (?e22 (concat ?e5 bvlambda_2_))
(let (?e23 (bvadd ?e21 ?e22))
(let (?e24 (extract[3:0] ?e23))
(let (?e25 (bvmul ?e7 ?e24))
(let (?e26 (extract[2:2] ?e25))
(let (?e27 (concat ?e2 ?e26))
(let (?e28 (bvadd ?e16 ?e27))
(let (?e29 (extract[1:1] ?e28))
(let (?e30 (concat ?e2 ?e29))
(let (?e31 (extract[3:3] ?e25))
(let (?e32 (concat ?e2 ?e31))
(let (?e33 (bvadd ?e30 ?e32))
(let (?e34 (extract[1:0] ?e33))
(let (?e35 (concat ?e6 ?e34))
(let (?e36 (extract[1:1] ?e14))
(let (?e37 (concat ?e2 ?e36))
(let (?e38 (bvadd ?e35 ?e37))
(let (?e39 (extract[0:0] ?e38))
(let (?e40 (extract[0:0] ?e28))
(let (?e41 (concat ?e39 ?e40))
(let (?e42 (extract[1:0] ?e25))
(let (?e43 (concat ?e41 ?e42))
(let (?e44 (ite (= ?e12 ?e43) bv1[1] bv0[1]))
(not (= (bvnot ?e44) bv0[1]))
)))))))))))))))))))))))))))))))))))))))))
```

Figure A.3: SMT formula defined in accordance with SMT-LIB version 1.2 [RT06] for property checking instance of (2×2) unsigned integer multiplier represented in Figure A.5

```
(benchmark mult_ub_2x2
:logic QF_BV
:extrafuns ((bvlambda_1_ BitVec[1]))
:extrafuns ((bvlambda_2_ BitVec[1]))
:extrafuns ((bvlambda_3_ BitVec[1]))
:extrafuns ((bvlambda_4_ BitVec[1]))
:formula
(let (?e1 bv0[2])
(let (?e2 bv0[1])
(let (?e7 (concat ?e1 bvlambda_3_))
(let (?e8 (concat ?e7 bvlambda_4_))
(let (?e9 (concat ?e1 bvlambda_1_))
(let (?e10 (concat ?e9 bvlambda_2_))
(let (?e11 (bvmul ?e8 ?e10))
(let (?e12 (concat ?e1 (bvnot bvlambda_1_)))
(let (?e13 (bvand bvlambda_2_ bvlambda_4_))
(let (?e14 (bvand bvlambda_1_ ?e13))
(let (?e15 (bvand (bvnot bvlambda_1_) (bvnot ?e13)))
(let (?e16 (bvand (bvnot ?e14) (bvnot ?e15)))
(let (?e17 (concat ?e1 ?e16))
(let (?e18 (bvadd ?e9 ?e17))
(let (?e19 (extract[1:1] ?e18))
(let (?e20 (concat ?e1 ?e19))
(let (?e21 (bvand bvlambda_2_ bvlambda_3_))
(let (?e22 (bvand bvlambda_1_ (bvnot bvlambda_2_)))
(let (?e23 (bvand bvlambda_4_ ?e22))
(let (?e24 (bvand (bvnot ?e21) (bvnot ?e23)))
(let (?e25 (ite (= bvlambda_1_ ?e24) bv1[1] bv0[1]))
(let (?e26 (concat ?e1 ?e25))
(let (?e27 (bvadd ?e20 ?e26))
(let (?e28 (extract[1:1] ?e27))
(let (?e29 (concat ?e1 ?e28))
(let (?e30 (bvand bvlambda_3_ ?e22))
(let (?e31 (bvand bvlambda_1_ ?e30))
(let (?e32 (bvand (bvnot bvlambda_1_) (bvnot ?e30)))
(let (?e33 (bvand (bvnot ?e31) (bvnot ?e32)))
(let (?e34 (concat ?e1 ?e33))
(let (?e35 (bvadd ?e29 ?e34))
(let (?e36 (extract[1:0] ?e35))
(let (?e37 (concat ?e2 ?e36))
(let (?e38 (bvand bvlambda_1_ bvlambda_4_))
(let (?e39 (concat ?e1 ?e38))
(let (?e40 (bvadd ?e37 ?e39))
(let (?e41 (extract[1:1] ?e40))
(let (?e42 (concat ?e1 ?e41))
(let (?e43 (bvadd ?e12 ?e42))
(let (?e44 (extract[1:0] ?e43))
(let (?e45 (concat ?e2 ?e44))
(let (?e46 (bvand bvlambda_1_ bvlambda_3_))
(let (?e47 (concat ?e1 ?e46))
(let (?e48 (bvadd ?e45 ?e47))
(let (?e49 (extract[0:0] ?e48))
(let (?e50 (extract[0:0] ?e40))
(let (?e51 (concat (bvnot ?e49) ?e50))
(let (?e52 (extract[0:0] ?e27))
(let (?e53 (concat ?e51 ?e52))
(let (?e54 (extract[0:0] ?e18))
(let (?e55 (concat ?e53 ?e54))
(let (?e56 (ite (= ?e11 ?e55) bv1[1] bv0[1]))
(not (= (bvnot ?e56) bv0[1]))
)))))))))))))))))))))))))))))))))))))))))))))))))))))
```

Figure A.4: SMT formula defined in accordance with SMT-LIB version 1.2 [RT06] for property checking instance of (2×2) unsigned integer multiplier represented in Figure A.6

```
// generated by:  ./gen_mult_adder -a 2 -b 2 -o mult_ub_2x2_abl.v -v 1 -e 2

// unsigned 2x2 Booth-encoded multiplier with primary inputs: a, b
// primary output: r


module mult_ub_2x2(r, a, b);

// The input-vectors of the circuit #0:
 input [1:0] a;
 input [1:0] b;

// The only output-vector:
 output [3:0] r;

// The partial products of the circuit #0:
 wire [3:0] pp_0_0;
 wire [1:0] pp_0_1;

// The adders of the circuit #0:
 wire [1:0] ha_0_0;
 wire [1:0] fa_0_0;

// *****  Partial products *****
 assign pp_0_0 = (-({b[1],1'b0}) + b[0])*a[1:0];
 assign pp_0_1 = b[1]*a[1:0];
// *****  Partial products *****

// *****  Adders *****
 assign ha_0_0 = ({1'b0,pp_0_1[0]} + {1'b0,pp_0_0[2]});
 assign fa_0_0 = ({1'b0,ha_0_0[1]} + {1'b0,pp_0_0[3]} + {1'b0,pp_0_1[1]});
// *****  Adders *****

// ***   outputs   ***
assign r[0] = pp_0_0[0];
assign r[1] = pp_0_0[1];
assign r[2] = ha_0_0[0];
assign r[3] = fa_0_0[0];

endmodule
```

Figure A.5: Verilog code of ($2{\times}2$) unsigned Booth-encoded integer multiplier at the ABL

```
// unsigned 2x2 Booth-encoded multiplier with primary inputs: a, b
// primary output: r


module mult_ub_2x2(r, a, b);

// The input-vectors of the circuit #0:
 input [1:0] a;
 input [1:0] b;

// The only output-vector:
 output [3:0] r;

// The adders of the circuit #0:
 wire [1:0] ha_0;
 wire [1:0] ha_1;
 wire [1:0] ha_2;
 wire [1:0] fa_0;
 wire [1:0] fa_1;

// *****  Control signals *****   Circuit #0
assign sh0_0_0 = b[0];
assign sh1_0_0 = ~b[0] & b[1];
assign cpl0_0 = b[1];
assign sh0_0_1 = b[1];
// sh1_0_1 is const_0
// cpl0_1 is const_0
// *****  Control signals *****   Circuit #0

// *****  Partial products *****
assign pp0_0_0 = (sh0_0_0 & a[0]) ^ cpl0_0;
assign pp0_1_0  = ((sh0_0_0 & a[1]) | (sh1_0_0 & a[0])) ^ cpl0_0;
assign pp0_2_0 = (sh1_0_0 & a[1]) ^ cpl0_0;
assign pp0_0_1 = sh0_0_1 & a[0];
assign nn0_0 = ~cpl0_0;
assign pp0_1_1 = sh0_0_1 & a[1];
// *****  Partial products *****

// *****  Adders *****
assign ha_0 = ({1'b0,cpl0_0} + {1'b0,pp0_0_0});
assign ha_1 = ({1'b0,ha_0[1]} + {1'b0,pp0_1_0});
assign fa_0 = ({1'b0,ha_1[1]} + {1'b0,pp0_2_0} + {1'b0,pp0_0_1});
assign fa_1 = ({1'b0,fa_0[1]} + {1'b0,nn0_0} + {1'b0,pp0_1_1});
assign ha_2 = ({1'b0,1'b1} + {1'b0,fa_1[0]});
// *****  Adders *****

// ***   outputs   ***
assign r[0] = ha_0[0];
assign r[1] = ha_1[0];
assign r[2] = fa_0[0];
assign r[3] = ha_2[0];

endmodule
```

Figure A.6: Verilog code of (2×2) unsigned integer multiplier with (radix-4) Booth-encoder described at the gate level and addition network described at the ABL

Appendix B

Complete Experimental Results for Suite of TriCore SMT Instances

Instance	CPU time, sec.	Instance	CPU time, sec.
th_madd_1_2.rtp.smt	23.8295	th_madd_2_2.rtp.smt	21.7054
th_madd_3_2.rtp.smt	25.7816	th_madd_4_2.rtp.smt	26.0336
th_madd_h1_2.rtp.smt	11.1487	th_madd_h1n1_2.rtp.smt	11.6047
th_madd_h1n1x8000hi_2.rtp.smt	7.9685	th_madd_h1n1x8000lo_2.rtp.smt	13.7929
th_madd_h2_2.rtp.smt	10.9047	th_madd_h2n1_2.rtp.smt	11.3847
th_madd_h2n1x8000hi_2.rtp.smt	7.81249	th_madd_h2n1x8000lo_2.rtp.smt	13.8209
th_madd_h3_2.rtp.smt	10.9527	th_madd_h3n1_2.rtp.smt	11.4287
th_madd_h3n1x8000hi_2.rtp.smt	7.47247	th_madd_h3n1x8000lo_2.rtp.smt	13.2728
th_madd_h4_2.rtp.smt	10.9647	th_madd_h4n1_2.rtp.smt	11.4047
th_madd_h4n1x8000hi_2.rtp.smt	7.28045	th_madd_h4n1x8000lo_2.rtp.smt	13.3888
th_madd_q1_2.rtp.smt	26.7657	th_madd_q1n1_2.rtp.smt	49.7951
th_madd_q2_2.rtp.smt	10.2326	th_madd_q2n1_2.rtp.smt	21.7774
th_madd_q3_2.rtp.smt	10.2486	th_madd_q3n1_2.rtp.smt	24.0975
th_madd_q4_2.rtp.smt	8.55253	th_madd_q4n1_2.rtp.smt	27.0897
th_madd_q4n1x8000lo_2.rtp.smt	17.7211	th_madd_q5_2.rtp.smt	8.39652
th_madd_q5n1_2.rtp.smt	23.3015	th_madd_q6_2.rtp.smt	25.9976
th_madd_q6n1_2.rtp.smt	49.9951	th_madd_q7_2.rtp.smt	39.4065
th_madd_q7n1_2.rtp.smt	10.5807	th_madd_q8_2.rtp.smt	38.3664
th_madd_q8n1_2.rtp.smt	10.6367	th_madd_q9_2.rtp.smt	8.52053
th_madd_q9n1_2.rtp.smt	30.5059	th_madd_q9n1x8000lo_2.rtp.smt	15.705
th_madd_q10_2.rtp.smt	8.13251	th_madd_q10n1_2.rtp.smt	21.6534
th_madd_u3_2.rtp.smt	25.2536	th_madd_u4_2.rtp.smt	24.5935
th_maddm_h1_2.rtp.smt	10.5567	th_maddm_h1n1_2.rtp.smt	25.2696
th_maddm_h1n1x8000hi_2.rtp.smt	27.7537	th_maddm_h1n1x8000lo_2.rtp.smt	16.297
th_maddm_h2_2.rtp.smt	10.5927	th_maddm_h2n1_2.rtp.smt	32.366
th_maddm_h2n1x8000hi_2.rtp.smt	25.1096	th_maddm_h2n1x8000lo_2.rtp.smt	16.389
th_maddm_h3_2.rtp.smt	10.5487	th_maddm_h3n1_2.rtp.smt	28.0538
th_maddm_h3n1x8000hi_2.rtp.smt	13.4368	th_maddm_h3n1x8000lo_2.rtp.smt	15.805
th_maddm_h4_2.rtp.smt	10.5287	th_maddm_h4n1_2.rtp.smt	32.614
th_maddm_h4n1x8000hi_2.rtp.smt	13.2408	th_maddm_h4n1x8000lo_2.rtp.smt	15.801
th_maddms_h5_2.rtp.smt	10.5807	th_maddms_h5n1_2.rtp.smt	35.4942
th_maddms_h5n1x8000hi_2.rtp.smt	28.7978	th_maddms_h5n1x8000lo_2.rtp.smt	16.145
th_maddms_h6_2.rtp.smt	10.2486	th_maddms_h6n1_2.rtp.smt	26.9537
th_maddms_h6n1x8000hi_2.rtp.smt	24.6655	th_maddms_h6n1x8000lo_2.rtp.smt	15.897
th_maddms_h7_2.rtp.smt	10.3606	th_maddms_h7n1_2.rtp.smt	23.1574
th_maddms_h7n1x8000hi_2.rtp.smt	77.7529	th_maddms_h7n1x8000lo_2.rtp.smt	99.0262
th_maddms_h8_2.rtp.smt	10.2486	th_maddms_h8n1_2.rtp.smt	38.3624
th_maddms_h8n1x8000hi_2.rtp.smt	78.3009	th_maddms_h8n1x8000lo_2.rtp.smt	98.1861

Table B.1: *TriCore* SMT formulas solved by *STABLE*

Instance	CPU time, sec.	Instance	CPU time, sec.
th_maddr_h1_2.rtp.smt	10.9527	th_maddr_h1n1_2.rtp.smt	11.3487
th_maddr_h1n1x8000hi_2.rtp.smt	7.82049	th_maddr_h1n1x8000lo_2.rtp.smt	15.749
th_maddr_h2_2.rtp.smt	10.8087	th_maddr_h2n1_2.rtp.smt	11.4887
th_maddr_h2n1x8000hi_2.rtp.smt	7.84449	th_maddr_h2n1x8000lo_2.rtp.smt	15.789
th_maddr_h3_2.rtp.smt	11.0367	th_maddr_h3n1_2.rtp.smt	11.5847
th_maddr_h3n1x8000hi_2.rtp.smt	7.28845	th_maddr_h3n1x8000lo_2.rtp.smt	85.5293
th_maddr_h4_2.rtp.smt	10.8687	th_maddr_h4n1_2.rtp.smt	11.3327
th_maddr_h4n1x8000hi_2.rtp.smt	7.31246	th_maddr_h4n1x8000lo_2.rtp.smt	86.8894
th_maddr_h5_2.rtp.smt	3.62423	th_maddr_h5n1_2.rtp.smt	11.7287
th_maddr_h5n1x8000hi_2.rtp.smt	7.86049	th_maddr_h5n1x8000lo_2.rtp.smt	15.497
th_maddr_q1_2.rtp.smt	8.86855	th_maddr_q1n1_2.rtp.smt	33.7861
th_maddr_q1n1x8000lo_2.rtp.smt	17.8251	th_maddr_q2_2.rtp.smt	8.79655
th_maddr_q2n1_2.rtp.smt	26.1136	th_maddrs_h6_2.rtp.smt	10.8967
th_maddrs_h6n1_2.rtp.smt	11.3527	th_maddrs_h6n1x8000hi_2.rtp.smt	7.74048
th_maddrs_h6n1x8000lo_2.rtp.smt	15.793	th_maddrs_h7_2.rtp.smt	11.0647
th_maddrs_h7n1_2.rtp.smt	11.5207	th_maddrs_h7n1x8000hi_2.rtp.smt	7.90049
th_maddrs_h7n1x8000lo_2.rtp.smt	15.929	th_maddrs_h8_2.rtp.smt	11.1807
th_maddrs_h8n1_2.rtp.smt	11.6287	th_maddrs_h8n1x8000hi_2.rtp.smt	7.28446
th_maddrs_h8n1x8000lo_2.rtp.smt	92.7218	th_maddrs_h9_2.rtp.smt	10.9247
th_maddrs_h9n1_2.rtp.smt	11.3967	th_maddrs_h9n1x8000hi_2.rtp.smt	7.32046
th_maddrs_h9n1x8000lo_2.rtp.smt	100.046	th_maddrs_h10_2.rtp.smt	11.2607
th_maddrs_h10n1_2.rtp.smt	11.6847	th_maddrs_h10n1x8000hi_2.rtp.smt	7.85649
th_maddrs_h10n1x8000lo_2.rtp.smt	15.489	th_maddrs_q3_2.rtp.smt	8.57654
th_maddrs_q3n1_2.rtp.smt	35.4142	th_maddrs_q3n1x8000lo_2.rtp.smt	16.349
th_maddrs_q4_2.rtp.smt	8.16051	th_maddrs_q4n1_2.rtp.smt	23.6135
th_madds_5_2.rtp.smt	30.5499	th_madds_6_2.rtp.smt	21.6334
th_madds_7_2.rtp.smt	26.0456	th_madds_8_2.rtp.smt	26.0136
th_madds_h5_2.rtp.smt	11.2007	th_madds_h5n1_2.rtp.smt	11.4567
th_madds_h5n1x8000hi_2.rtp.smt	7.9845	th_madds_h5n1x8000lo_2.rtp.smt	13.8329
th_madds_h6_2.rtp.smt	11.1327	th_madds_h6n1_2.rtp.smt	11.3847
th_madds_h6n1x8000hi_2.rtp.smt	7.79249	th_madds_h6n1x8000lo_2.rtp.smt	13.8129
th_madds_h7_2.rtp.smt	11.1447	th_madds_h7n1_2.rtp.smt	11.3887
th_madds_h7n1x8000hi_2.rtp.smt	7.50847	th_madds_h7n1x8000lo_2.rtp.smt	13.3048
th_madds_h8_2.rtp.smt	11.1487	th_madds_h8n1_2.rtp.smt	11.5807
th_madds_h8n1x8000hi_2.rtp.smt	7.31246	th_madds_h8n1x8000lo_2.rtp.smt	13.3448
th_madds_q11_2.rtp.smt	27.3897	th_madds_q11n1_2.rtp.smt	50.5272
th_madds_q12_2.rtp.smt	10.2606	th_madds_q12n1_2.rtp.smt	30.5579
th_madds_q13_2.rtp.smt	10.2166	th_madds_q13n1_2.rtp.smt	31.862
th_madds_q14_2.rtp.smt	8.43253	th_madds_q14n1_2.rtp.smt	21.7374
th_madds_q14n1x8000lo_2.rtp.smt	15.717	th_madds_q15_2.rtp.smt	8.38452
th_madds_q15n1_2.rtp.smt	26.4417	th_madds_q16_2.rtp.smt	25.9976
th_madds_q16n1_2.rtp.smt	48.9471	th_madds_q17_2.rtp.smt	10.4967
th_madds_q17n1_2.rtp.smt	10.5207	th_madds_q18_2.rtp.smt	35.4422
th_madds_q18n1_2.rtp.smt	10.5527	th_madds_q19_2.rtp.smt	8.47253
th_madds_q19n1_2.rtp.smt	26.7497	th_madds_q19n1x8000lo_2.rtp.smt	15.729
th_madds_q20_2.rtp.smt	8.09651	th_madds_q20n1_2.rtp.smt	28.5898
th_madds_u5_2.rtp.smt	25.3336	th_madds_u6_2.rtp.smt	22.1174
th_madds_u7_2.rtp.smt	25.2176	th_madds_u8_2.rtp.smt	22.3894
th_maddsu_h1_2.rtp.smt	15.0729	th_maddsu_h1n1_2.rtp.smt	37.7984
th_maddsu_h1n1x8000hi_2.rtp.smt	10.9847	th_maddsu_h1n1x8000lo_2.rtp.smt	15.889
th_maddsu_h2_2.rtp.smt	15.1929	th_maddsu_h2n1_2.rtp.smt	35.9862
th_maddsu_h2n1x8000hi_2.rtp.smt	10.9207	th_maddsu_h2n1x8000lo_2.rtp.smt	15.769
th_maddsu_h3_2.rtp.smt	15.0609	th_maddsu_h3n1_2.rtp.smt	34.4302
th_maddsu_h3n1x8000hi_2.rtp.smt	8.62854	th_maddsu_h3n1x8000lo_2.rtp.smt	72.0885
th_maddsu_h4_2.rtp.smt	15.1889	th_maddsu_h4n1_2.rtp.smt	27.8377
th_maddsu_h4n1x8000hi_2.rtp.smt	10.9647	th_maddsu_h4n1x8000lo_2.rtp.smt	71.1004
th_maddsum_h1_2.rtp.smt	21.8454	th_maddsum_h1n1_2.rtp.smt	41.6546
th_maddsum_h1n1x8000hi_2.rtp.smt	30.8299	th_maddsum_h1n1x8000lo_2.rtp.smt	21.8374
th_maddsum_h2_2.rtp.smt	25.4736	th_maddsum_h2n1_2.rtp.smt	45.8389
th_maddsum_h2n1x8000hi_2.rtp.smt	32.9261	th_maddsum_h2n1x8000lo_2.rtp.smt	22.3654
th_maddsum_h3_2.rtp.smt	25.8136	th_maddsum_h3n1_2.rtp.smt	59.9597
th_maddsum_h3n1x8000hi_2.rtp.smt	136.197	th_maddsum_h3n1x8000lo_2.rtp.smt	22.9494

Table B.2: *TriCore* SMT formulas solved by *STABLE*, cont.

Instance	CPU time, sec.	Instance	CPU time, sec.
th_maddsum_h4_2.rtp.smt	25.1096	th_maddsum_h4n1_2.rtp.smt	41.0466
th_maddsum_h4n1x8000hi_2.rtp.smt	80.373	th_maddsum_h4n1x8000lo_2.rtp.smt	128.408
th_maddsums_h5_2.rtp.smt	21.2413	th_maddsums_h5n1_2.rtp.smt	43.6987
th_maddsums_h5n1x8000hi_2.rtp.smt	35.8222	th_maddsums_h5n1x8000lo_2.rtp.smt	21.1373
th_maddsums_h6_2.rtp.smt	21.2493	th_maddsums_h6n1_2.rtp.smt	30.0059
th_maddsums_h6n1x8000hi_2.rtp.smt	33.0541	th_maddsums_h6n1x8000lo_2.rtp.smt	25.1056
th_maddsums_h7_2.rtp.smt	21.1693	th_maddsums_h7n1_2.rtp.smt	29.3418
th_maddsums_h7n1x8000hi_2.rtp.smt	114.163	th_maddsums_h7n1x8000lo_2.rtp.smt	86.7494
th_maddsums_h8_2.rtp.smt	21.4213	th_maddsums_h8n1_2.rtp.smt	30.1539
th_maddsums_h8n1x8000hi_2.rtp.smt	94.6939	th_maddsums_h8n1x8000lo_2.rtp.smt	125.408
th_maddsur_h1_2.rtp.smt	14.8889	th_maddsur_h1n1_2.rtp.smt	30.5539
th_maddsur_h1n1x8000hi_2.rtp.smt	10.6767	th_maddsur_h1n1x8000lo_2.rtp.smt	16.109
th_maddsur_h2_2.rtp.smt	15.1369	th_maddsur_h2n1_2.rtp.smt	35.6062
th_maddsur_h2n1x8000hi_2.rtp.smt	10.8207	th_maddsur_h2n1x8000lo_2.rtp.smt	16.169
th_maddsur_h3_2.rtp.smt	15.253	th_maddsur_h3n1_2.rtp.smt	35.1582
th_maddsur_h3n1x8000hi_2.rtp.smt	8.29652	th_maddsur_h3n1x8000lo_2.rtp.smt	71.6285
th_maddsur_h4_2.rtp.smt	14.9609	th_maddsur_h4n1_2.rtp.smt	38.1984
th_maddsur_h4n1x8000hi_2.rtp.smt	8.48853	th_maddsur_h4n1x8000lo_2.rtp.smt	83.5372
th_maddsurs_h5_2.rtp.smt	14.9049	th_maddsurs_h5n1_2.rtp.smt	33.2101
th_maddsurs_h5n1x8000hi_2.rtp.smt	10.6967	th_maddsurs_h5n1x8000lo_2.rtp.smt	16.109
th_maddsurs_h6_2.rtp.smt	15.0889	th_maddsurs_h6n1_2.rtp.smt	32.31
th_maddsurs_h6n1x8000hi_2.rtp.smt	10.7807	th_maddsurs_h6n1x8000lo_2.rtp.smt	16.073
th_maddsurs_h7_2.rtp.smt	15.1969	th_maddsurs_h7n1_2.rtp.smt	25.4776
th_maddsurs_h7n1x8000hi_2.rtp.smt	8.33652	th_maddsurs_h7n1x8000lo_2.rtp.smt	79.385
th_maddsurs_h8_2.rtp.smt	15.1849	th_maddsurs_h8n1_2.rtp.smt	37.4823
th_maddsurs_h8n1x8000hi_2.rtp.smt	8.42853	th_maddsurs_h8n1x8000lo_2.rtp.smt	79.0969
th_maddsus_h5_2.rtp.smt	15.0449	th_maddsus_h5n1_2.rtp.smt	35.6742
th_maddsus_h5n1x8000hi_2.rtp.smt	11.0247	th_maddsus_h5n1x8000lo_2.rtp.smt	15.853
th_maddsus_h6_2.rtp.smt	15.1209	th_maddsus_h6n1_2.rtp.smt	37.7624
th_maddsus_h6n1x8000hi_2.rtp.smt	10.7927	th_maddsus_h6n1x8000lo_2.rtp.smt	15.761
th_maddsus_h7_2.rtp.smt	15.373	th_maddsus_h7n1_2.rtp.smt	29.8579
th_maddsus_h7n1x8000hi_2.rtp.smt	8.64854	th_maddsus_h7n1x8000lo_2.rtp.smt	87.6575
th_maddsus_h8_2.rtp.smt	15.377	th_maddsus_h8n1_2.rtp.smt	34.5062
th_maddsus_h8n1x8000hi_2.rtp.smt	8.36852	th_maddsus_h8n1x8000lo_2.rtp.smt	77.0568
th_msub_1_2.rtp.smt	32.558	th_msub_2_2.rtp.smt	28.5898
th_msub_3_2.rtp.smt	73.8846	th_msub_4_2.rtp.smt	74.2526
th_msub_h1_2.rtp.smt	19.2172	th_msub_h1n1_2.rtp.smt	43.6587
th_msub_h1n1x8000hi_2.rtp.smt	11.1087	th_msub_h1n1x8000lo_2.rtp.smt	18.7772
th_msub_h2_2.rtp.smt	19.1092	th_msub_h2n1_2.rtp.smt	39.7505
th_msub_h2n1x8000hi_2.rtp.smt	11.0447	th_msub_h2n1x8000lo_2.rtp.smt	18.8612
th_msub_h3_2.rtp.smt	19.2292	th_msub_h3n1_2.rtp.smt	28.4738
th_msub_h3n1x8000hi_2.rtp.smt	8.70054	th_msub_h3n1x8000lo_2.rtp.smt	86.9094
th_msub_h4_2.rtp.smt	19.2652	th_msub_h4n1_2.rtp.smt	39.8185
th_msub_h4n1x8000hi_2.rtp.smt	8.60854	th_msub_h4n1x8000lo_2.rtp.smt	79.901
th_msub_q1_2.rtp.smt	17.1371	th_msub_q1n1_2.rtp.smt	17.0011
th_msub_q2_2.rtp.smt	24.0735	th_msub_q2n1_2.rtp.smt	45.1028
th_msub_q2n1tru32_2.rtp.smt	23.4775	th_msub_q2tru32_2.rtp.smt	24.4415
th_msub_q3_2.rtp.smt	24.1575	th_msub_q3n1_2.rtp.smt	42.2346
th_msub_q3n1tru32_2.rtp.smt	23.5215	th_msub_q3tru32_2.rtp.smt	24.5855
th_msub_q4_2.rtp.smt	13.7249	th_msub_q4n1_2.rtp.smt	29.1298
th_msub_q4n1x8000lo_2.rtp.smt	20.8013	th_msub_q5_2.rtp.smt	13.1208
th_msub_q5n1_2.rtp.smt	23.2575	th_msub_q6_2.rtp.smt	76.6888
th_msub_q6n1_2.rtp.smt	99.9182	th_msub_q7_2.rtp.smt	25.9896
th_msub_q7n1_2.rtp.smt	38.4784	th_msub_q8_2.rtp.smt	26.2016
th_msub_q8n1_2.rtp.smt	40.3465	th_msub_q9_2.rtp.smt	20.2533
th_msub_q9n1_2.rtp.smt	37.4543	th_msub_q9n1x8000lo_2.rtp.smt	26.0536
th_msub_q10_2.rtp.smt	19.2452	th_msub_q10n1_2.rtp.smt	41.0786
th_msub_u3_2.rtp.smt	48.551	th_msub_u4_2.rtp.smt	48.579

Table B.3: *TriCore* SMT formulas solved by *STABLE*, cont.

Instance	CPU time, sec.	Instance	CPU time, sec.
th_msubad_h1_2.rtp.smt	15.1649	th_msubad_h1n1_2.rtp.smt	14.9969
th_msubad_h1n1x8000hi_2.rtp.smt	8.0925	th_msubad_h1n1x8000lo_2.rtp.smt	18.5772
th_msubad_h2_2.rtp.smt	15.1489	th_msubad_h2n1_2.rtp.smt	15.0369
th_msubad_h2n1x8000hi_2.rtp.smt	8.0365	th_msubad_h2n1x8000lo_2.rtp.smt	18.5892
th_msubad_h3_2.rtp.smt	15.1529	th_msubad_h3n1_2.rtp.smt	14.9849
th_msubad_h3n1x8000hi_2.rtp.smt	7.61247	th_msubad_h3n1x8000lo_2.rtp.smt	75.5487
th_msubad_h4_2.rtp.smt	15.1449	th_msubad_h4n1_2.rtp.smt	15.0649
th_msubad_h4n1x8000hi_2.rtp.smt	7.51647	th_msubad_h4n1x8000lo_2.rtp.smt	81.9291
th_msubadm_h1_2.rtp.smt	14.0009	th_msubadm_h1n1_2.rtp.smt	37.0943
th_msubadm_h1n1x8000hi_2.rtp.smt	28.0858	th_msubadm_h1n1x8000lo_2.rtp.smt	18.9052
th_msubadm_h2_2.rtp.smt	14.0609	th_msubadm_h2n1_2.rtp.smt	30.6819
th_msubadm_h2n1x8000hi_2.rtp.smt	26.3216	th_msubadm_h2n1x8000lo_2.rtp.smt	18.9212
th_msubadm_h3_2.rtp.smt	14.1169	th_msubadm_h3n1_2.rtp.smt	29.4858
th_msubadm_h3n1x8000hi_2.rtp.smt	133.608	th_msubadm_h3n1x8000lo_2.rtp.smt	136.969
th_msubadm_h4_2.rtp.smt	14.1129	th_msubadm_h4n1_2.rtp.smt	32.9461
th_msubadm_h4n1x8000hi_2.rtp.smt	133.712	th_msubadm_h4n1x8000lo_2.rtp.smt	136.993
th_msubadms_h5_2.rtp.smt	14.1769	th_msubadms_h5n1_2.rtp.smt	28.2378
th_msubadms_h5n1x8000hi_2.rtp.smt	25.2816	th_msubadms_h5n1x8000lo_2.rtp.smt	19.0172
th_msubadms_h6_2.rtp.smt	14.0969	th_msubadms_h6n1_2.rtp.smt	35.7342
th_msubadms_h6n1x8000hi_2.rtp.smt	31.45	th_msubadms_h6n1x8000lo_2.rtp.smt	18.9732
th_msubadms_h7_2.rtp.smt	14.1889	th_msubadms_h7n1_2.rtp.smt	27.8337
th_msubadms_h7n1x8000hi_2.rtp.smt	121.684	th_msubadms_h7n1x8000lo_2.rtp.smt	136.949
th_msubadms_h8_2.rtp.smt	14.0889	th_msubadms_h8n1_2.rtp.smt	46.3069
th_msubadms_h8n1x8000hi_2.rtp.smt	126.504	th_msubadms_h8n1x8000lo_2.rtp.smt	136.965
th_msubadr_h1_2.rtp.smt	14.6929	th_msubadr_h1n1_2.rtp.smt	14.4449
th_msubadr_h1n1x8000hi_2.rtp.smt	7.86449	th_msubadr_h1n1x8000lo_2.rtp.smt	18.6812
th_msubadr_h2_2.rtp.smt	14.8369	th_msubadr_h2n1_2.rtp.smt	14.8889
th_msubadr_h2n1x8000hi_2.rtp.smt	7.9285	th_msubadr_h2n1x8000lo_2.rtp.smt	18.6092
th_msubadr_h3_2.rtp.smt	14.8129	th_msubadr_h3n1_2.rtp.smt	14.7489
th_msubadr_h3n1x8000hi_2.rtp.smt	7.37646	th_msubadr_h3n1x8000lo_2.rtp.smt	77.8729
th_msubadr_h4_2.rtp.smt	14.6409	th_msubadr_h4n1_2.rtp.smt	14.6649
th_msubadr_h4n1x8000hi_2.rtp.smt	7.37646	th_msubadr_h4n1x8000lo_2.rtp.smt	93.5658
th_msubadrs_h5_2.rtp.smt	14.5729	th_msubadrs_h5n1_2.rtp.smt	14.4249
th_msubadrs_h5n1x8000hi_2.rtp.smt	7.86049	th_msubadrs_h5n1x8000lo_2.rtp.smt	18.6892
th_msubadrs_h6_2.rtp.smt	14.8729	th_msubadrs_h6n1_2.rtp.smt	14.7849
th_msubadrs_h6n1x8000hi_2.rtp.smt	8.0005	th_msubadrs_h6n1x8000lo_2.rtp.smt	18.7812
th_msubadrs_h7_2.rtp.smt	14.9529	th_msubadrs_h7n1_2.rtp.smt	14.6969
th_msubadrs_h7n1x8000hi_2.rtp.smt	7.34046	th_msubadrs_h7n1x8000lo_2.rtp.smt	74.2886
th_msubadrs_h8_2.rtp.smt	14.8729	th_msubadrs_h8n1_2.rtp.smt	14.8649
th_msubadrs_h8n1x8000hi_2.rtp.smt	7.36846	th_msubadrs_h8n1x8000lo_2.rtp.smt	91.4057
th_msubads_h5_2.rtp.smt	14.8009	th_msubads_h5n1_2.rtp.smt	14.7049
th_msubads_h5n1x8000hi_2.rtp.smt	0	th_msubads_h5n1x8000lo_2.rtp.smt	18.4572
th_msubads_h6_2.rtp.smt	14.7849	th_msubads_h6n1_2.rtp.smt	14.9409
th_msubads_h6n1x8000hi_2.rtp.smt	8.0525	th_msubads_h6n1x8000lo_2.rtp.smt	18.4972
th_msubads_h7_2.rtp.smt	15.0729	th_msubads_h7n1_2.rtp.smt	14.8569
th_msubads_h7n1x8000hi_2.rtp.smt	0	th_msubads_h7n1x8000lo_2.rtp.smt	0.004
th_msubads_h8_2.rtp.smt	14.8129	th_msubads_h8n1_2.rtp.smt	14.7929
th_msubads_h8n1x8000hi_2.rtp.smt	7.53647	th_msubads_h8n1x8000lo_2.rtp.smt	78.5889
th_msubm_h1_2.rtp.smt	28.5818	th_msubm_h1n1_2.rtp.smt	47.363
th_msubm_h1n1x8000hi_2.rtp.smt	33.7581	th_msubm_h1n1x8000lo_2.rtp.smt	25.4936
th_msubm_h2_2.rtp.smt	29.7019	th_msubm_h2n1_2.rtp.smt	50.9352
th_msubm_h2n1x8000hi_2.rtp.smt	31.874	th_msubm_h2n1x8000lo_2.rtp.smt	24.5095
th_msubm_h3_2.rtp.smt	28.7418	th_msubm_h3n1_2.rtp.smt	38.7864
th_msubm_h3n1x8000hi_2.rtp.smt	136.593	th_msubm_h3n1x8000lo_2.rtp.smt	141.689
th_msubm_h4_2.rtp.smt	28.8698	th_msubm_h4n1_2.rtp.smt	45.3108
th_msubm_h4n1x8000hi_2.rtp.smt	136.357	th_msubm_h4n1x8000lo_2.rtp.smt	141.381
th_msubms_h5_2.rtp.smt	29.3898	th_msubms_h5n1_2.rtp.smt	37.4663
th_msubms_h5n1x8000hi_2.rtp.smt	32.234	th_msubms_h5n1x8000lo_2.rtp.smt	25.8856
th_msubms_h6_2.rtp.smt	29.4418	th_msubms_h6n1_2.rtp.smt	44.3788
th_msubms_h6n1x8000hi_2.rtp.smt	33.3061	th_msubms_h6n1x8000lo_2.rtp.smt	24.4935

Table B.4: *TriCore* SMT formulas solved by *STABLE*, cont.

Instance	CPU time, sec.	Instance	CPU time, sec.
th_msubms_h7_2.rtp.smt	28.4698	th_msubms_h7n1_2.rtp.smt	45.2228
th_msubms_h7n1x8000hi_2.rtp.smt	138.165	th_msubms_h7n1x8000lo_2.rtp.smt	141.229
th_msubms_h8_2.rtp.smt	28.6458	th_msubms_h8n1_2.rtp.smt	39.5305
th_msubms_h8n1x8000hi_2.rtp.smt	137.241	th_msubms_h8n1x8000lo_2.rtp.smt	141.385
th_msubr_h1_2.rtp.smt	18.7412	th_msubr_h1n1_2.rtp.smt	39.2505
th_msubr_h1n1x8000hi_2.rtp.smt	10.7767	th_msubr_h1n1x8000lo_2.rtp.smt	18.9692
th_msubr_h2_2.rtp.smt	18.9332	th_msubr_h2n1_2.rtp.smt	32.49
th_msubr_h2n1x8000hi_2.rtp.smt	10.7847	th_msubr_h2n1x8000lo_2.rtp.smt	18.9332
th_msubr_h3_2.rtp.smt	18.7572	th_msubr_h3n1_2.rtp.smt	37.0183
th_msubr_h3n1x8000hi_2.rtp.smt	8.41653	th_msubr_h3n1x8000lo_2.rtp.smt	88.5335
th_msubr_h4_2.rtp.smt	18.7412	th_msubr_h4n1_2.rtp.smt	29.2778
th_msubr_h4n1x8000hi_2.rtp.smt	8.39252	th_msubr_h4n1x8000lo_2.rtp.smt	83.5732
th_msubr_h5_2.rtp.smt	18.9892	th_msubr_h5n1_2.rtp.smt	32.506
th_msubr_h5n1x8000hi_2.rtp.smt	10.8927	th_msubr_h5n1x8000lo_2.rtp.smt	18.7212
th_msubr_q1_2.rtp.smt	13.4808	th_msubr_q1n1_2.rtp.smt	30.7459
th_msubr_q1n1x8000lo_2.rtp.smt	23.9135	th_msubr_q2_2.rtp.smt	13.0448
th_msubr_q2n1_2.rtp.smt	28.9018	th_msubrs_h6_2.rtp.smt	18.7732
th_msubrs_h6n1_2.rtp.smt	32.8021	th_msubrs_h6n1x8000hi_2.rtp.smt	10.7407
th_msubrs_h6n1x8000lo_2.rtp.smt	18.9652	th_msubrs_h7_2.rtp.smt	18.7132
th_msubrs_h7n1_2.rtp.smt	32.542	th_msubrs_h7n1x8000hi_2.rtp.smt	10.7407
th_msubrs_h7n1x8000lo_2.rtp.smt	18.9332	th_msubrs_h8_2.rtp.smt	18.6812
th_msubrs_h8n1_2.rtp.smt	37.0623	th_msubrs_h8n1x8000hi_2.rtp.smt	8.39653
th_msubrs_h8n1x8000lo_2.rtp.smt	88.9536	th_msubrs_h9_2.rtp.smt	18.7212
th_msubrs_h9n1_2.rtp.smt	29.5698	th_msubrs_h9n1x8000hi_2.rtp.smt	8.38452
th_msubrs_h9n1x8000lo_2.rtp.smt	83.9892	th_msubrs_h10_2.rtp.smt	19.0692
th_msubrs_h10n1_2.rtp.smt	33.5781	th_msubrs_h10n1x8000hi_2.rtp.smt	10.9127
th_msubrs_h10n1x8000lo_2.rtp.smt	18.8052	th_msubrs_q3_2.rtp.smt	13.5968
th_msubrs_q3n1_2.rtp.smt	21.3773	th_msubrs_q3n1x8000lo_2.rtp.smt	20.9133
th_msubrs_q4_2.rtp.smt	13.1008	th_msubrs_q4n1_2.rtp.smt	20.8533
th_msubs_5_2.rtp.smt	72.6725	th_msubs_6_2.rtp.smt	52.7673
th_msubs_7_2.rtp.smt	73.5926	th_msubs_8_2.rtp.smt	52.0273
th_msubs_h5_2.rtp.smt	19.3812	th_msubs_h5n1_2.rtp.smt	30.1739
th_msubs_h5n1x8000hi_2.rtp.smt	11.1487	th_msubs_h5n1x8000lo_2.rtp.smt	18.9292
th_msubs_h6_2.rtp.smt	19.1812	th_msubs_h6n1_2.rtp.smt	38.1024
th_msubs_h6n1x8000hi_2.rtp.smt	11.1007	th_msubs_h6n1x8000lo_2.rtp.smt	18.9172
th_msubs_h7_2.rtp.smt	19.3652	th_msubs_h7n1_2.rtp.smt	45.1228
th_msubs_h7n1x8000hi_2.rtp.smt	8.60454	th_msubs_h7n1x8000lo_2.rtp.smt	70.7684
th_msubs_h8_2.rtp.smt	19.3932	th_msubs_h8n1_2.rtp.smt	41.3546
th_msubs_h8n1x8000hi_2.rtp.smt	8.58854	th_msubs_h8n1x8000lo_2.rtp.smt	86.4014
th_msubs_q11_2.rtp.smt	77.4008	th_msubs_q11n1_2.rtp.smt	104.087
th_msubs_q12_2.rtp.smt	24.4895	th_msubs_q12n1_2.rtp.smt	24.2215
th_msubs_q12n1tru32_2.rtp.smt	24.2135	th_msubs_q12tru32_2.rtp.smt	24.6935
th_msubs_q13_2.rtp.smt	20.5413	th_msubs_q13n1_2.rtp.smt	38.8744
th_msubs_q13n1tru32_2.rtp.smt	23.4495	th_msubs_q13tru32_2.rtp.smt	20.6253
th_msubs_q14_2.rtp.smt	13.6289	th_msubs_q14n1_2.rtp.smt	35.9862
th_msubs_q14n1x8000lo_2.rtp.smt	20.7813	th_msubs_q15_2.rtp.smt	13.1408
th_msubs_q15n1_2.rtp.smt	30.7419	th_msubs_q16_2.rtp.smt	77.1448
th_msubs_q16n1_2.rtp.smt	103.77	th_msubs_q17_2.rtp.smt	25.8616
th_msubs_q17n1_2.rtp.smt	39.1264	th_msubs_q18_2.rtp.smt	25.2536
th_msubs_q18n1_2.rtp.smt	34.5982	th_msubs_q19_2.rtp.smt	20.0173
th_msubs_q19n1_2.rtp.smt	35.2262	th_msubs_q19n1x8000lo_2.rtp.smt	26.3216
th_msubs_q20_2.rtp.smt	19.0132	th_msubs_q20n1_2.rtp.smt	49.7431
th_msubs_u5_2.rtp.smt	47.979	th_msubs_u6_2.rtp.smt	48.295
th_msubs_u7_2.rtp.smt	47.959	th_msubs_u8_2.rtp.smt	48.187
th_mul_1_2.rtp.smt	27.1777	th_mul_2_2.rtp.smt	20.6373
th_mul_3_2.rtp.smt	23.1214	th_mul_4_2.rtp.smt	24.6655
th_mul_h1_2.rtp.smt	24.5055	th_mul_h1n1_2.rtp.smt	11.1527
th_mul_h1n1x8000hi_2.rtp.smt	7.32846	th_mul_h1n1x8000lo_2.rtp.smt	8.28052
th_mul_h2_2.rtp.smt	24.5095	th_mul_h2n1_2.rtp.smt	11.1967
th_mul_h2n1x8000hi_2.rtp.smt	7.30046	th_mul_h2n1x8000lo_2.rtp.smt	8.24851

Table B.5: *TriCore* SMT formulas solved by *STABLE*, cont.

Instance	CPU time, sec.	Instance	CPU time, sec.
th_mul_h3_2.rtp.smt	24.5135	th_mul_h3n1_2.rtp.smt	11.7607
th_mul_h3n1x8000hi_2.rtp.smt	7.00844	th_mul_h3n1x8000lo_2.rtp.smt	8.0205
th_mul_h4_2.rtp.smt	24.5135	th_mul_h4n1_2.rtp.smt	11.2247
th_mul_h4n1x8000hi_2.rtp.smt	6.99244	th_mul_h4n1x8000lo_2.rtp.smt	8.41253
th_mul_q1_2.rtp.smt	6.57241	th_mul_q1n1_2.rtp.smt	10.5087
th_mul_q1n1x8000lo_2.rtp.smt	10.9447	th_mul_q2_2.rtp.smt	5.52435
th_mul_q2n1_2.rtp.smt	6.4684	th_mul_q3_2.rtp.smt	8.0605
th_mul_q3n1_2.rtp.smt	8.16451	th_mul_q4_2.rtp.smt	8.0765
th_mul_q4n1_2.rtp.smt	7.48447	th_mul_q5_2.rtp.smt	25.6776
th_mul_q5n1_2.rtp.smt	26.3656	th_mul_q6_2.rtp.smt	9.73261
th_mul_q6n1_2.rtp.smt	9.76861	th_mul_q7_2.rtp.smt	8.29652
th_mul_q7n1_2.rtp.smt	8.38852	th_mul_q8_2.rtp.smt	25.7696
th_mul_q8n1_2.rtp.smt	26.1376	th_mul_u3_2.rtp.smt	24.2775
th_mul_u4_2.rtp.smt	23.5415	th_mulm_h1_2.rtp.smt	10.5847
th_mulm_h1n1_2.rtp.smt	10.6087	th_mulm_h1n1x8000hi_2.rtp.smt	8.64454
th_mulm_h1n1x8000lo_2.rtp.smt	9.38059	th_mulm_h2_2.rtp.smt	10.0486
th_mulm_h2n1_2.rtp.smt	10.0886	th_mulm_h2n1x8000hi_2.rtp.smt	8.14451
th_mulm_h2n1x8000lo_2.rtp.smt	8.90056	th_mulm_h3_2.rtp.smt	10.0526
th_mulm_h3n1_2.rtp.smt	10.0406	th_mulm_h3n1x8000hi_2.rtp.smt	8.40853
th_mulm_h3n1x8000lo_2.rtp.smt	9.02056	th_mulm_h4_2.rtp.smt	10.6767
th_mulm_h4n1_2.rtp.smt	10.7127	th_mulm_h4n1x8000hi_2.rtp.smt	8.24451
th_mulm_h4n1x8000lo_2.rtp.smt	8.91256	th_mulms_h5_2.rtp.smt	9.32858
th_mulms_h5n1_2.rtp.smt	9.37659	th_mulms_h5n1x8000hi_2.rtp.smt	7.24845
th_mulms_h5n1x8000lo_2.rtp.smt	8.45653	th_mulms_h6_2.rtp.smt	9.87262
th_mulms_h6n1_2.rtp.smt	9.96462	th_mulms_h6n1x8000hi_2.rtp.smt	7.79249
th_mulms_h6n1x8000lo_2.rtp.smt	8.88856	th_mulms_h7_2.rtp.smt	9.81261
th_mulms_h7n1_2.rtp.smt	9.91662	th_mulms_h7n1x8000hi_2.rtp.smt	7.66448
th_mulms_h7n1x8000lo_2.rtp.smt	8.24851	th_mulms_h8_2.rtp.smt	9.79261
th_mulms_h8n1_2.rtp.smt	9.79261	th_mulms_h8n1x8000hi_2.rtp.smt	7.33646
th_mulms_h8n1x8000lo_2.rtp.smt	8.62054	th_mulr_h1_2.rtp.smt	10.2766
th_mulr_h1n1_2.rtp.smt	11.8847	th_mulr_h1n1x8000hi_2.rtp.smt	7.34846
th_mulr_h1n1x8000lo_2.rtp.smt	8.82455	th_mulr_h2_2.rtp.smt	10.7687
th_mulr_h2n1_2.rtp.smt	11.9047	th_mulr_h2n1x8000hi_2.rtp.smt	7.39246
th_mulr_h2n1x8000lo_2.rtp.smt	8.87255	th_mulr_h3_2.rtp.smt	10.8767
th_mulr_h3n1_2.rtp.smt	11.9887	th_mulr_h3n1x8000hi_2.rtp.smt	7.07644
th_mulr_h3n1x8000lo_2.rtp.smt	8.54453	th_mulr_h4_2.rtp.smt	10.3286
th_mulr_h4n1_2.rtp.smt	11.3887	th_mulr_h4n1x8000hi_2.rtp.smt	6.62841
th_mulr_h4n1x8000lo_2.rtp.smt	8.11651	th_mulr_q1_2.rtp.smt	6.07638
th_mulr_q1n1_2.rtp.smt	9.91662	th_mulr_q1n1x8000lo_2.rtp.smt	11.0487
th_mulr_q2_2.rtp.smt	9.40859	th_mulr_q2n1_2.rtp.smt	9.50859
th_muls_5_2.rtp.smt	25.2376	th_muls_6_2.rtp.smt	25.2456
th_muls_u5_2.rtp.smt	22.4614	th_muls_u6_2.rtp.smt	21.5253

Table B.6: *TriCore* SMT formulas solved by *STABLE*, cont.

List of Figures

List of Tables

Bibliography

[ABC$^+$02] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A SAT-based approach for solving formulas over boolean and linear mathematical propositions. In *Proc. International Conference on Automated Deduction (CAD)*, pages 195–210, 2002.

[AF07] Bijan Alizadeh and Masahiro Fujita. M.: LTED: A Canonical and Compact Hybrid Word-Boolean Representation as a Formal Model for Hardware/Software Co-designs. In *The fourth Workshop on Constraints in Formal Verification (CFV 2007)*, pages 15–29, 2007.

[AF08] Bijan Alizadeh and Masahiro Fujita. Modular-HED: A canonical decision diagram for modular equivalence verification of polynomial functions. In *CFV08*, Australia, September 2008.

[AIS86] IEEE Standard for Logic Circuit Diagrams. Corrected Edition. *ANSI/IEEE Std 991-1986*, 1986.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[AL03] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases*. (Graduate studies in mathematics) AMS, 2003.

[And02] Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[AS06] Jaakko T. Astola and Radomir S. Stankovic. *Fundamentals of Switching Theory and Logic Design*. Springer, 2006.

[Ash01] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[Bar77] Jon Barwise. *Handbook of mathematical logic*. North-Holland Publ., Amsterdam [u.a.], 1977.

[BB09] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.

[BBC$^+$05] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *Journal of Automated Reasoning*, 35(1–3):265–293, October 2005.

[BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 535–541, New York, NY, USA, 1995. ACM.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999.

[BCF$^+$07] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In *CAV*, pages 547–560, 2007.

[BCL$^+$94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[BD94] Bernd Becker and Rolf Drechsler. OFDD Based Minimization of Fixed Polarity Reed-Muller Expressions Using Hybrid Genetic Algorithms. In *ICCD*, pages 106–110. IEEE Computer Society, 1994.

[BD09] Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326–1345, 2009. Effective Methods in Algebraic Geometry.

[BDG$^+$09] Michael Brickenstein, Alexander Dreyer, Gert-Martin Greuel, Markus Wedler, and Oliver Wienand. New developments in the theory of Gröbner bases and applications to formal verification. *Journal of Pure and Applied Algebra*, 213:1612–1635, 2009.

[BDL98] Clark. W. Barrett, David. L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proc. International Design Automation Conference (DAC)*, 1998.

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 236–249, London, UK, 2002. Springer-Verlag.

[BDT95] Bernd Becker, Rolf Drechsler, and Michael Theobald. OKFDDs versus OBDDs and OFDDs. In Zoltán Fülöp and Ferenc Gécseg, editors, *ICALP*, volume 944 of *Lecture Notes in Computer Science*, pages 475–486. Springer, 1995.

[BH08] Domagoj Babić and Frank Hutter. Spear Theorem Prover. In *Proc. of the SAT 2008 Race*, 2008.

[Bha99] J. Bhasker. *A Verilog HDL Primer, Second Edition*. Star Galaxy Publishing, 1999.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[BJW04] R. Brinkmann, P. Johannsen, and K. Winkelmann. Application of property checking and underlying techniques. Boston, MA, USA, 2004. Kluwer Academic Publishers.

[BKO$^+$07] Al E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, Bryan Brady, and Eth Zürich. Deciding bit-vector arithmetic with abstraction. In *In Proc. TACAS 2007*, pages 358–372. Springer, 2007.

[boo] boolector. `http://fmv.jku.at/boolector`.

[BPST10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer, 2010.

[BR96] Stephen Brown and Jonathan Rose. FPGA and CPLD architectures: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.

[Bra93] Daniel Brand. Verification of large synthesized designs. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 534–537, 1993.

[Bro90] F. M. Brown. *Boolean Reasoning (The Logic of Boolean Equations)*. Kluwer Academic Publishers, 1990.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Volume 185 of Biere et al. [BHvMW09], February 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[Buc76] B. Buchberger. Some properties of gröbner-bases for polynomial ideals. *SIGSAM Bull.*, 10(4):19–24, 1976.

[CE81] E. M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science*, 131, 1981.

[CFM$^+$93] E. M. Clarke, M. Fujita, P. McGeer, K. L. McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. In *Proc. International Workshop on Logic Synthesis*, pages (P6a) 1–15, 1993.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, London, England, 1999.

[CK03] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proc. International Design Automation Conference (DAC)*, pages 830–835, 2003.

[CLO07] David A. Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[CZKR02] M. Ciesielski, Z. Zeng, P. Kalla, and B. Rouzeyre. Taylor expansion diagrams: A compact, canonical representation with applications to symbolic verification. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, pages 285–291, 2002.

[DB98a] Rolf Drechsler and Bernd Becker. *Graphenbasierte Funktionsdarstellung - Boolesche und Pseudo-Boolesche Funktionen*. Leitfäden der Informatik. Teubner, 1998.

[DB98b] Rolf Drechsler and Bernd Becker. Ordered Kronecker functional decision diagrams - a data structure for representation and manipulation of Boolean functions. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(10):965–973, 1998.

[DBJ98] Rolf Drechsler, Bernd Becker, and Andrea Jahnke. On Variable Ordering and Decomposition Type Choice in OKFDDs. *IEEE Trans. Computers*, 47(12):1398–1403, 1998.

[DBR96] R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs: A new data structure for verification. In *Proc. European Design and Test Conference (EDTC)*, pages 2–8, 1996.

[DBS$^+$94] Rolf Drechsler, Bernd Becker, A. Sarabi, M. Theobald, and M. Perkowski. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Proc. International Design Automation Conference (DAC)*, pages 415–419, 1994.

[DdM06] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. International Conference Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DM06] Bruno Dutertre and Leonardo De Moura. The yices SMT solver. Technical report, 2006.

[dMB07] Leonardo Mendonça de Moura and Nikolaj Bjoerner. Efficient E-Matching for SMT Solvers. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[EBD] EBDDRES. `http://fmv.jku.at/ebddres`.

[ES03] N. Een and N. Soerensson. An extensible SAT-solver. In *Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT)*, May 2003.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the Computer Aided Verification Conference*, pages 524–536. Springer, 2007.

[GHB01] Wolfgang Günther, Andreas Hett, and Bernd Becker. Application of linearly transformed BDDs in sequential verification. In *Proc. Asia and South Pacific Design Automation Conference, 2001*, pages 91–96, 2001.

[GHN$^+$04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 26–37, July 2004.

[GN02] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust SAT solver. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[GP07] G.-M. Greuel and G. Pfister. *A SINGULAR Introduction to Commutative Algebra*. Springer Verlag, Berlin, Heidelberg, New York, 2nd edition, 2007. 705 pages.

[GPS10] G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 3-1-2 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de`, 2010.

[GSW10] G. M. Greuel, F. Seelisch, and O. Wienand. The Groebner basis of the ideal of vanishing polynomials. *Journal of Symbolic Computation*, page 14, To be published 2010.

[GT96] W. K. Grassmann and J.-P. Tremblay. *Logic and Discrete Mathematics*. Prentice Hall, 1996.

[HCCG96] S.Y. Huang, K.T. Cheng, K.C. Chen, and Uwe Glaeser. An ATPG-based framework for verifying sequential equivalence. In *Proc. International Test Conference (ITC)*, 1996.

[HMY95] Kiyoharu Hamaguchi, Akihito Morita, and Shuzo Yajima. Efficient construction of binary moment diagrams for verifying arithmetic circuits. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 78–82, November 1995.

[HP02] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[HS96] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, 1996.

[Hu97] Alan J. Hu. Formal Hardware Verification with BDDs: An Introduction. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, pages 677–682, 1997.

[Inf] Infineon Technologies AG. Tricore 2 architectural manual, doc v1.0. `http://www.infineon.com/tricore`.

[Int] Intel Pentium FDIV bug. `http://www.cs.earlham.edu/~dusko/cs63/fdiv.html`.

[JLS09] Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. 2009.

[JMF95] J. Jain, R. Mukherjee, and M. Fujita. Advanced verification techniques based on learning. In *Proc. International Design Automation Conference (DAC)*, pages 420 – 426, 1995.

[JSB06] Toni Jussila, Carsten Sinz, and Armin Biere. Extended resolution proofs for symbolic sat solving with quantification. In *In: Proc. of SAT. LNCS 4121*, pages 54–60. Springer, 2006.

[KDB$^+$03] Martin Keim, Rolf Drechsler, Bernd Becker, Michael Martin, and Paul Molitor. Polynomial formal verification of multipliers. *Formal Methods in System Design*, 22(1):39–58, 2003.

[KF02] M. Kubo and M. Fujita. Debug methodology for arithmetic circuits on FPGAs. pages 236 – 242, dec. 2002.

[KJW$^+$08] U. Krautz, C. Jacobi, K. Weber, M. Pflanz, W. Kunz, and M. Wedler. Verifying full-custom multipliers by boolean equivalence checking and an arithmetic bit level proof. In *ASP-DAC '08: Proceedings of the 2008 conference on Asia and South Pacific design automation*, pages 398–403, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[KK97] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Proc. International Design Automation Conference (DAC)*, pages 263–268, November 1997.

[Kor98] I. Koren. *Computer Arithmetic Algorithms*. Brooside Court Publishers, 1998.

[KS97] W. Kunz and D. Stoffel. *Reasoning in Boolean Networks - Logic Synthesis and Verification Using Testing Techniques*. Kluwer Academic Publishers, Boston, 1997.

[KS07] D. Kroening and S. A. Seshia. Formal verification at higher levels of abstraction. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, 2007.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[KSR92] U. Kebschull, E. Schubert, and W. Rostenstiel. Multi-level logic based on functional decision diagrams. In *Proc. European Design Automation Conference (EDAC)*, pages 43–47, 1992.

[Kun93] W. Kunz. An efficient tool for logic verification based on recursive learning. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 538–543, November 1993.

[Lee59] C. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, July 1959.

[LS95] Yung-Te Lai and Sarama Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. International Design Automation Conference (DAC)*, pages 254–260, 1995.

[Mat96] Y. Matsunaga. An efficient equivalence checker for combinational circuits. In *Proc. International Design Automation Conference (DAC)*, pages 629–634, June 1996.

[MB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[Mic94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[Min93] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC '93: Proceedings of the 30th international Design Automation Conference*, pages 272–277, New York, NY, USA, 1993. ACM.

[Mis01] Alan Mishchenko. An introduction to zero-suppressed binary decision diagrams. Technical report, in Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, 2001.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff engineering an efficient SAT solver. In *Proc. International Design Automation Conference (DAC)*, 2001.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MSS99] P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Decision procedures for SAT, SAT Modulo Theories and beyond. The BarcelogicTools. In *LPAR'05, LNCS 3835*, pages 23–46. Springer, 2005.

[NSWK05] M. D. Nguyen, D. Stoffel, M. Wedler, and W. Kunz. Transition-by-transition FSM traversal for reachability analysis in bounded model checking. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, San Jose, USA, November 2005.

[NTM$^+$09] M.D. Nguyen, M. Thalmaier, M.Wedler, D. Stoffel, and W. Kunz. A re-use methodology for formal soc protocol compliance. In *Proc. Forum on Specification & Design Languages(FDL)*, Sophia Antipolis, France, September 2009.

[NTW$^+$08] Minh D. Nguyen, Max Thalmaier, Markus Wedler, Jörg Bormann, Dominik Stoffel, and Wolfgang Kunz. Unbounded protocol compliance verification using interval property checking with invariants. *IEEE Transactions on Computer-Aided Design*, 27(11):2068–2082, November 2008.

[One] Onespin Solutions GmbH. Germany. OneSpin 360MV. `http://www.onespin-solutions.com`.

[Pre] PrecoSAT 236. `http://fmv.jku.at/precosat/`.

[PWS$^+$08] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, O. Wienand, and E. Karibaev. Modeling of Custom-Designed Arithmetic Components for ABL Normalization. In *GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2008.

[PWS$^+$11] Evgeny Pavlenko, Markus Wedler, Dominik Stoffel, Wolfgang Kunz, Frank Seelisch, Gert-Martin Greuel, and Alexander Dreyer. STABLE: A new QF-BV SMT solver for hard verification problems combining Boolean reasoning with computer algebra. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, 2011.

[PWSK07] E. Pavlenko, M. Wedler, D. Stoffel, and W. Kunz. Arithmetic Constrains in SAT-based Property Checking. In *10. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2007.

[RT06] Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2006.

[SAF08] O. Sarbishei, B. Alizadeh, and M. Fujita. Arithmetic circuits verification without looking for internal equivalences. pages 7 –16, jun. 2008.

[SB01] Christoph Scholl and Bernd Becker. Checking equivalence for partial implementations. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 238–243, New York, NY, USA, 2001. ACM.

[SB06] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining bdds. In *In: Proc. of the 1st Intl. Computer Science Symp. in Russia (CSR 2006). LNCS 3967*, pages 600–611. Springer, 2006.

[SBW98] C. Scholl, B. Becker, and T.M. Weis. Word-level decision diagrams, WLCDs and division. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 672–677, 1998.

[Sha38] C. Shannon. A symbolic analysis of relay and switching circuits. *Transactions AIEE*, 57:713–723, 1938.

[SK97] D. Stoffel and W. Kunz. Record & play a structural fixed point iteration for sequential circuit verification. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 394–399, November 1997.

[SK01] D. Stoffel and W. Kunz. Verification of integer multipliers on the arithmetic bit level. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 183–189, San Jose, CA, November 2001.

[SK04] D. Stoffel and W. Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *IEEE Transactions on Computer-Aided Design*, 23(5):586–597, May 2004.

[SKE06] N. Shekhar, P. Kalla, and F. Enescu. Equivalence verification of arithmetic datapath with multiple word-length operands. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, 2006.

[SKE07] Namrata Shekhar, Priyank Kalla, and Florian Enescu. Equivalence verification of polynomial datapaths using ideal membership testing. *IEEE Transactions on Computer-Aided Design*, 26(7):1320–1330, July 2007.

[SKEG05] Namrata Shekhar, Priyank Kalla, Florian Enescu, and Sivaram Gopalakrishnan. Equivalence verification of polynomial datapaths with fixed-size bit-vectors using finite ring algebra. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, 2005.

[SKME08] Namrata Shekhar, Priyank Kalla, M. Brandon Meredith, and Florian Enescu. Simulation bounds for equivalence verification of polynomial datapaths using finite ring algebra. *IEEE Trans. Very Large Scale Integr. Syst.*, 16:376–387, April 2008.

[SLB03] Sanjit A. Seshia, Shuvendu K. Lahiri, and Randal E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proc. International Design Automation Conference*, 2003.

[SMT07] SMT-COMP 2007. `http://www.smtcomp.org/2007/`, 2007.

[SMT08] SMT-COMP 2008. `http://www.smtcomp.org/2008/`, 2008.

[SMT09] SMT-COMP 2009. `http://www.smtcomp.org/2009/`, 2009.

[SMT10] SMT-COMP 2010. `http://www.smtcomp.org/2010/`, 2010.

[Spe] Spear. `http://www.domagoj-babic.com`.

[STAF09] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fujita. A formal approach for debugging arithmetic circuits. *IEEE Transactions on Computer-Aided Design*, 28(05):742–754, May 2009.

[STP] STP. `http://sites.google.com/site/stpfastprover/STP-Fast-Prover`.

[SWWK04] D. Stoffel, M. Wedler, P. Warkentin, and W. Kunz. Structural FSM-traversal. *IEEE Transactions on Computer-Aided Design*, 23(5):598–619, May 2004.

[Tse68] G. S. Tseitin. *On the complexity of proof in prepositional calculus*, volume 8 of *Studies in constructive mathematics and mathematical logic. Part II*, pages 234–259. "Nauka", Leningrad. Otdel., Leningrad, 1968.

[vE98] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, pages 618–623, Paris, France, March 1998.

[VVSA07] Shobha Vasudevan, Vinod Viswanath, Robert W. Sumners, and Jacob A.Abraham. Automatic verification of arithmetic circuits in rtl using stepwise refinement of term rewriting systems. *IEEE Transactions on Computers*, 56(10):1401–1414, 2007.

[web] Excerpts from a Conversation with Gordon Moore: Moore's Law. `ftp://download.intel.com/museum/Moores_Law/Video-transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf`.

[WHAH07] Yuki Watanabe, Naofumi Homma, Takafumi Aoki, and Tatsuo Higuchi. Application of symbolic computer algebra to arithmetic circuit verification. In *Proc. International Conference on Computer Design (ICCD)*, pages 25 – 32, October 2007.

[Wie] O. Wienand. Standard Bases over Ring and Applications. PhD Thesis. To appear in 2011.

[WM00] Sandro Wefel and Paul Molitor. Prove that a faulty multiplier is faulty!? In *GLSVLSI '00: Proceedings of the 10th Great Lakes symposium on VLSI*, pages 43–46, New York, NY, USA, 2000. ACM Press.

[WPD$^+$10] Markus Wedler, Evgeny Pavlenko, Alexander Dreyer, Frank Seelisch, Dominik Stoffel, Gert-Martin Greuel, and Wolfgang Kunz. Solving Hard Instances in QF-BV Combining Boolean Reasoning with Computer Algebra. In *Algorithms and Applications for Next Generation SAT Solvers*, number 09461 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[WSBK07] M. Wedler, D. Stoffel, R. Brinkmann, and W. Kunz. A normalization method for arithmetic data-path verification. *IEEE Transactions on Computer-Aided Design*, 26(11):1909–1922, November 2007.

[WSK04] M. Wedler, D. Stoffel, and W. Kunz. Arithmetic reasoning in DPLL-based SAT solving. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, Paris, France, February 2004.

[WSK05] M. Wedler, D. Stoffel, and W. Kunz. Normalization at the arithmetic bit level. In *Proc. International Design Automation Conference (DAC-05)*, June 2005.

[WWS$^+$08] Oliver Wienand, Markus Wedler, Dominik Stoffel, Wolfgang Kunz, and Gert-Martin Greuel. An algebraic approach for proving data correctness in arithmetic data paths. In *Proc. International Conference Computer Aided Verification (CAV)*, pages 473–486, Princeton, NJ, USA, July 2008.

[Yic] Yices. `http://yices.csl.sri.com/`.

[Z3] Z3. `http://research.microsoft.com/en-us/um/redmond/projects/z3/index.html`.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 279–285, 2001.

Lebenslauf

Name:	Evgeny Pavlenko
Geburtsort:	Atbasar, Kasachstan

Ausbildung

09/1986 - 06/1997	Schulbidung Atbasar, Kasachstan
09/1997 - 06/2001	Bachelor Polytechnische Universität Tomsk, Russland
07/2001 - 06/2002	Diplom Polytechnische Universität Tomsk, Russland
04/2004 - 09/2006	Master Technische Universität Kaiserslautern
10/2006 - 12/2011	Promotion Technische Universität Kaiserslautern

Berufstätigkeit

10/2002 - 02/2004	Ingenieur Messtechnik “LipezkElektroKommunikation” AG, Lipezk, Russland
04/2004 - 06/2011	Wissenschaftliche Hilfskraft Technische Universität Kaiserslautern Lehrstuhl Entwurf Informationstechnischer Systeme