July 2015

Function Verification of Combinational Arithmetic Circuits

Duo Liu

University of Massachusetts Amherst

FUNCTION VERIFICATION OF COMBINATIONAL ARITHMETIC CIRCUITS

A Thesis Presented

by

DUO LIU

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

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

May 2015

Electrical and Computer Engineering

FUNCTION VERIFICATION OF COMBINATIONAL ARITHMETIC CIRCUITS

A Thesis Presented

by

DUO LIU

ACKNOWLEDGMENTS

I would like to thank Professor. Ciesielski who gives me the opportunity to dedicate myself into this important research and guides me throughout my study in UMASS Amherst. I also would like to thank Walter for developing very useful functions that I used in my software. Special thanks to Cunxi who wrote the benchmark generator BenGen. I have used unsigned multipliers generated by BenGen in the experiments. Thanks to Arun and Sudarshan for answering my questions and friends outside the VLSI CAD lab who help me with my research. Thank you my dear parents, your selfless supports make me understand the meaning of love.

ABSTRACT

FUNCTION VERIFICATION OF COMBINATIONAL ARITHMETIC CIRCUIT

MAY 2015

DUO LIU

B.S., JIANGNAN UNIVERSITY, WUXI, JIANGSU, CHINA

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

Directed by: Professor Maciej Ciesielski

Hardware design verification is the most challenging part in overall hardware design process. It is because design size and complexity are growing very fast while the requirement for performance is ever higher. Conventional simulation-based verification method cannot keep up with the rapid increase in the design size, since it is impossible to exhaustively test all input vectors of a complex design. An important part of hardware verification is combinational arithmetic circuit verification. It draws a lot of attention because flattening the design into bit-level, known as the bit-blasting problem, hinders the efficiency of many current formal techniques. The goal of this thesis is to introduce a robust and efficient formal verification method for combinational integer arithmetic circuit based on an in-depth analysis of recent advances in computer algebra. The method proposed here solves the verification problem at bit level, while avoiding bit-blasting problem. It also avoids the expensive Groebner basis computation, typically employed by symbolic computer algebra methods.

The proposed method verifies the gate-level implementation of the design by representing the design components (logic gates and arithmetic modules) by polynomials in $\mathbb{Z}_{2^n}$. It then transforms the polynomial representing the output bits (called "output signature") into a unique polynomial in input signals (called "input signature") using gate-level information of the design. The computed input signature is then compared with the reference input signature (golden model) to determine whether the circuit behaves as anticipated. If the reference input signature is not given, our method can be used to compute (or extract) the arithmetic function of the design by computing its input signature. Additional tools, based on canonical word-level design representations (such as TED or BMD) can be used to determine the function of the computed input signature represents. We demonstrate the applicability of the proposed method to arithmetic circuit verification on a large number of designs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

1.1 Verification and Its Importance

Digital hardware verification is becoming more and more challenging. It is because design scale and functionality are growing very fast while the requirement for performance is even higher. It is painful and costly to redesign the circuit if bugs are found very late in the design process. Verification is the one technology for designers to assure the reliance, accuracy and functionality of designs at early stage of work flow.

Validation and verification are two basic techniques to demonstrate that a design is correct. Validation checks if the design's specification meets the market's needs, it is typically done using simulation. Verification checks if the design meets its specifications. We only focus on verification in this thesis.

Verification process tries to make sure a design works exactly as the designer anticipated. It is a process that penetrates modern circuit design. Figure 1.1 shows a complete VLSI design flow. This figure shows that most of the total design time, from Register Transfer Level (RTL) to logic level, is consumed by verification. More specifically, more than 70 percent of the design time and resources are spent on functional verification on average [1]. Despite all these efforts, functional bugs still force companies to redesign their products. An important reason for this situation is the limitations in current verification methods which will be introduced in the following sections.

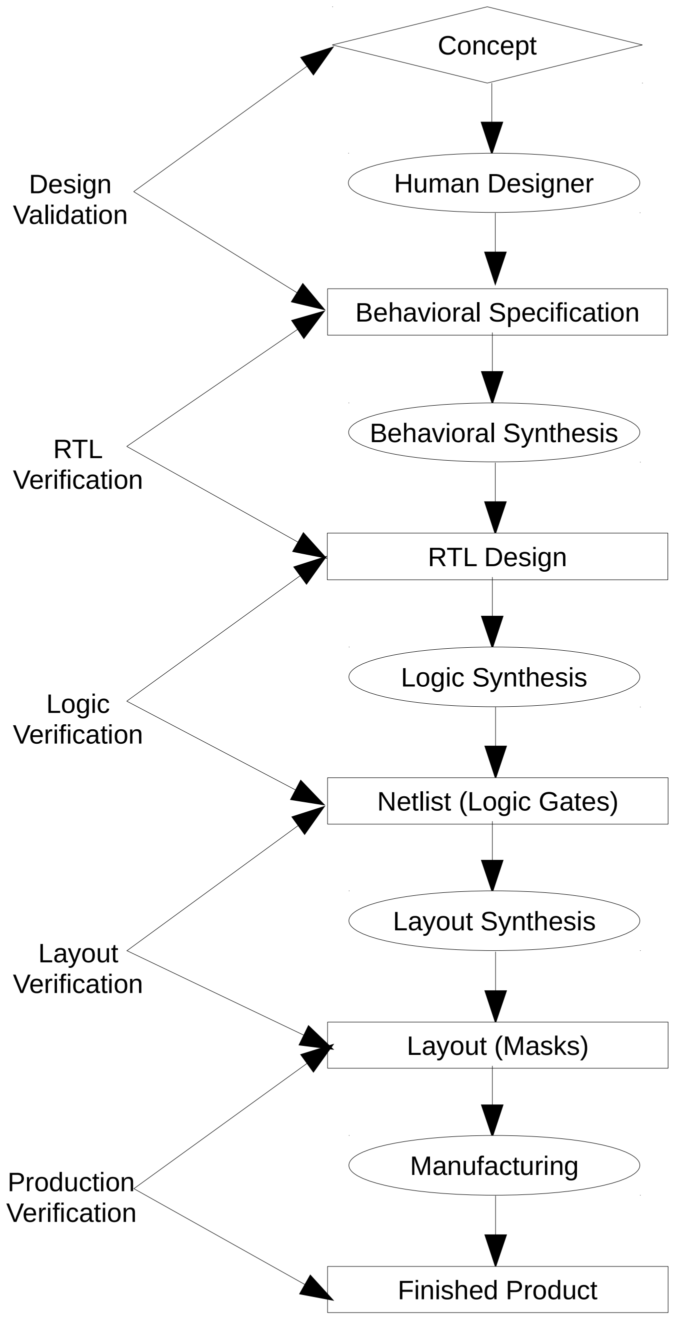


Figure 1.1: VLSI Design Flow [1].

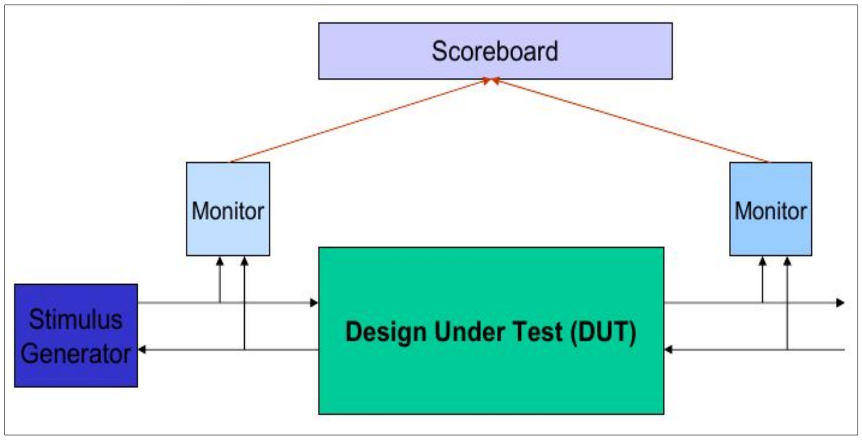


Figure 1.2: Simulation-Based Verification [2].

1.2 Simulation-Based Verification

The traditional way to test design functionality is to simulate the designed circuits function before tape out. To perform simulation of a design, one loads the design into a simulator, assigns a sequence of input vectors and criterion to the simulator and then runs the simulator to check if the circuit behaves as expected the under the given input stimulus. This process is shown in Figure 1.2. In this figure, the scoreboard checks design behavior and the monitors sample interface activity. Through a simulation process, one can design and debug a dynamic model of an actual system either for the purpose of understanding the system behavior or evaluating various strategies (like constraints or optimizations) for the operation of the system [3].

However, with the ever-increasing size and complexity of integrated circuits and systems on chip (SoC) [4], it is becoming harder and harder to simulate the Design Under Test (DUT). The number of problem cases to be examined increase dramatically for larger and more complex designs. Designers try to put all these cases into a test vector file. Typically this step cannot be done satisfactorily because there are cases that designers may never think of. Without exhausting all possible cases, it is very likely that some corner cases in disguise will be omitted during simulation. Even if all possible cases are considered by designer, it is still impossible to run through all these cases within a reasonable length of time because of the huge quantity of possible test cases. In other words, just as Edsgar W. Dijkstra said, “Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence [5]”.

1.3 Formal Verification

Taking the above factors into consideration, formal verification can be seen as a good complement for simulation-based verification considering that completeness is the greatest advantage of formal verification. It has already been proved that formal verification methods can be successfully applied to combinational arithmetic circuits [6], [7], [8]. In this section, a series of methods which aim at implementing and improving formal verification methodologies are presented. These works display many important attempts which are pervasively used in formal verification solutions nowadays.

Functional formal verification discussed here is called “formal” because formal methods of mathematics are used to prove that a design implements the correct function. It is a precise technique in the sense of its completeness. Formal verification can guarantee the functional correctness of the design [9] with high confidence, if used correctly, and avoids tremendous cost of fixing bugs that come to surface late in the whole design chain. Formal verification uses mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness [10]. In the process of implementing formal verification for a design, mathematical models of the system implementation and of the specification must be built first as a formal description of the function of the design. *Specification* in formal verification defines the desired behavior or properties of the system, while *implementation* represents how the circuit is constructed in detail. Then, based on the established mathematical implementation model and specification description, engineers use mathematical reasoning to verify whether design intent (specification) is preserved in the implementation [11]. The current popular formal verification methods include equivalence checking, symbolic simulation, model checking, theorem proving, ATPG (Automatic Test Pattern Generation), and others.

1.3.1 Equivalence Checking

Generally speaking, equivalence checking investigates whether two given expressions are functionally identical. In the hardware verification area, equivalence checking plays the role of proving or disproving that a pair of circuit designs behave exactly the same. Typically, the circuit which is known to be correct is called *reference* while the other one is the *implementation* of the DUT. Equivalence checking allows the user to find, analyze and eliminate all the errors introduced during the transfer from one level of abstraction to another [12]. It is often used when engineering changes are made before final manufacturing. Engineering changes are manual corrections made in the implementation process if design errors are found. However, these changes themselves are likely to introduce new errors. Equivalence checking ensures the function of a design after correction is as intended. Besides, appending optimization and testing circuitry, such as self-scan logic and power-control circuitry, to original designs also may change the design function without being noticed. In this case, the change will require fixing the bugs.

There are many ways to do equivalence checking. Instead of using simulation, which has shortcomings discussed before, people are more inclined to identify similarities between the structures of the designs. Identifying structural similarities means to find functionally equivalent internal nodes in the pair of designs to be compared. Mathematical methods are used frequently to accomplish this identification task. Most of them are based on canonical data structures, such as Binary Decision Diagrams (BDDs) [6], Binary Moment Diagrams (BMDs) [7], Taylor Expansion Diagrams (TEDs) [13] and their extensions [14]. Concrete methods that implement equivalence checking will be illustrated in the following chapter.

1.3.2 Model Checking

Model checking, or property checking, has been proposed as a hardware verification method over 30 years ago [15]. It can be a good complement to equivalence checking, specifically for finite state concurrent systems. For example, it can be used to ascertain whether two circuits which are judged to have different functions after equivalence checking are really different. This is useful in practice because in equivalence checking, all too often, designs of different levels need to be compared. Examples include comparing an RTL design with a gate-level design to check their equivalence. In such cases, the lower level design might contain structural details that the higher level design does not. Due to this, equivalence checking might report a false negative.

In basic model checking, there are mainly three integral parts: implementation, specification and model checker. The implementation (or system model) is represented as a state-transition graph that shows transition relations between different states. The specification represents the property that must be satisfied. Specifications are expressed in prepositional temporal logic form, a form for time-dependent Boolean functions. The model checker is a module that exhaustively searches the implementation's state space to find as many states that satisfy the specification as possible. Figure 1.3 shows the work flow of a general model checking system.

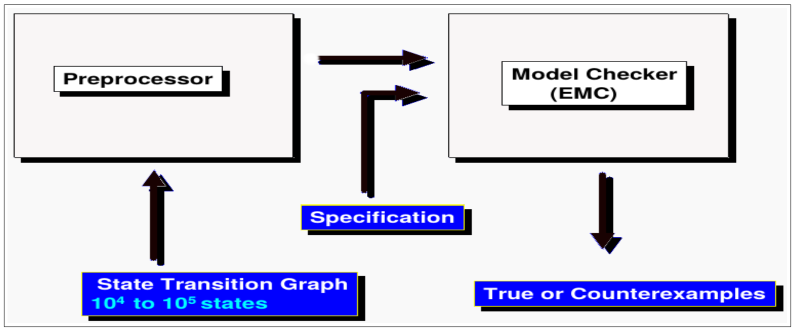


Figure 1.3: Model checking example [16].

An alternative way to implement model checking is to model both the implementation and the specification as automata. Then, two automata can be compared to test equivalence.

An advantage of model checking is that if the model checker fails to prove the equivalence between implementation and specification, it will generate counterexamples which give clues why the model checking fails and where the errors might be.

The success of the model checker depends on the correct formulation of the specification and the power of the model checker. The Hardware Model Checking Competition (HWMCC) is held every year to test the efficiency of model checkers. Although model checking is faster than theorem proving (see Section 1.3.4) and is completely automatic, it faces state explosion problem when dealing with larger designs. The difficulty of automata comparison is another factor that limit the pervasive application of model checking.

1.3.3 Symbolic Simulation

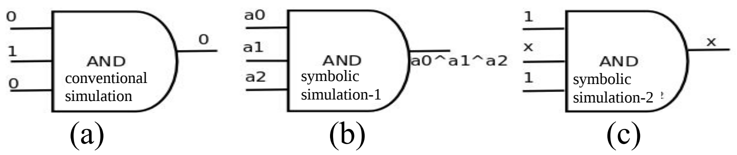


Figure 1.4: An example of conventional simulation and symbolic simulations.

Symbolic simulation "involves introducing an expanded set of signal values and redefining the basic simulation functions to operate over this expanded set", according to [17]. In symbolic simulation, each signal is represented by a symbolic value which can take any value in the symbolic value domain. For example, in Boolean domain, one symbolic value can be regarded as wither binary 0 or binary 1. During symbolic simulation, the symbols are propagated through the design from primary inputs to primary outputs. This content is illustrated in Figure 1.4. The sign "$\wedge$" in Figure 1.4 means logical *AND*. Figure 1.4 (a) represents conventional simulation which guarantees the correctness of only one case in each run. Figure 1.4 (b) represents a complete symbolic simulation. Since *a0*, *a1* and *a2* are symbolic binary values, the output of one run actually contains the information provided by 8 conventional simulation runs. Figure

1.4 (c) represents partial symbolic simulation. By assigning symbolic value to one of the inputs and performing constant propagation, the number of total simulation runs can also be reduced. The significant reduction in simulation cases renders symbolic simulation ideal for handling complex and large circuits that otherwise need huge amount of conventional simulation runs. Figure 1.5 shows a more complex example.

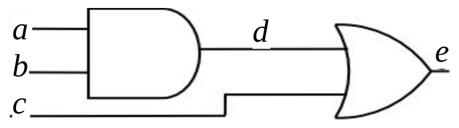


Figure 1.5: Another example of symbolic simulation

In Figure 1.5, symbols *a*, *b* and *c* are primary inputs, symbol *d* is intermediate signal and symbol *e* is primary output. Signal d can be expressed as $d = a \wedge b$. Then *e* can be represented as $e = (a \wedge b) \vee c$. All primary input symbols propagate to the primary output in one simulation run. The primary output bit is represented by a symbolic expression with respect to primary inputs, according to specific circuit.

While performing symbolic simulation, one needs to make sure that the initial state and input variables cover all valid test cases within circuit constraints [18]. BDDs are popular in digital circuit symbolic simulation [17] because they can represent sets of values that signals in circuit may take with reasonable complexity. However, BDDs are not feasible for large designs because certain internal structures, such as XOR, can make BDDs too large to build.

Based on basic symbolic simulation theory, a set of improved works have been proposed for hardware verification, such as ternary simulation [19], quaternary simulation, and symbolic trajectory simulation [20].

1.3.4 Theorem Proving

Theorem proving is another important technique used for hardware verification. It is of great importance because of the need for more general-purpose mathematic theorems and tools that can be applied to verification. Designers need more general mathematics because as hardware design improves, the mathematical models for integer and binary areas are not sufficient for solving all verification problems. Some verification problems for certain designs, such as floating-point arithmetic circuits, need mathematical models for more complex fields, such as infinite sets and real numbers.

The first step in theorem proving is to build a collection of formulas derived from the circuit under the proper field. The overall formulation process includes sub-tasks, such as defining semantics and syntax and formulating specification (or conjecture). Then the satisfiability between derived formulas and formulated specification is checked using the selected theory. Some typical formalisms that are common in theorem proving are propositional logic, temporal logic, first-order logic and higher-order logic. The critical factors that determine the selection of formalism type are the formula expressiveness and the difficulty of solving corresponding decision problem. More expressive formalisms will be harder to automate. In practice, it has been found that propositional logic is suitable for modeling a wide range of problems, such as combinational logic equivalence checking problems and finite-state transition problems. First-order logic is also powerful enough for modeling current problems.

After selecting feasible fields, proper theorem proving techniques, such as resolution, tableaux, and others, are applied to check whether the specification theoretically follows from inferences derived from the circuitry.

There are important differences between model checking and theorem proving. Model checking converts system and specification into certain models (usually state transition graphs) and then check their equivalence. Theorem proving models specification as conjecture and system as axioms, then tries to prove the conjecture using known axioms and theories in the assumed field.

Other works propose a combination of simulation and formal verification. These methods are typically based on Brand's work [21] where a divide and conquer paradigm is introduced. Firstly a small number of simulations are run on both designs to ferret out possible equivalent points. Then techniques, such as comparing ROBDDs or SAT sweeping can be used to prove that these points are indeed equivalent, [22], [23]. Subsequently, the proved true equivalences between subset areas in the designs are used to deduct further equivalences of the circuits until the whole designs are explored. However, approaches that follow such framework also have some problems. The biggest one is the possibility of false negatives [24], and for this reason they can't always claim that two designs which are proved different using this method are indeed different.

1.3.5 BDD-Based Techniques

Many of the formal verification methods make use of Binary Decision Diagrams (BDDs) [25], [26]. Being an effective data structure to present Boolean functions, BDDs are popular in formal verification area, where circuit functional verification problems can be described in a Boolean fashion and efficiently solved using BDD-based techniques.

To construct BDDs for boolean functions correctly, some rules must be followed. Firstly, a fixed ordering of variables which appear in the given function specification must be specified. Then the initial decision tree is built from root to leaves according to the predefined ordering. The Boolean-Shannon decomposition rule used while building the decision tree is:

$$F = xF_x + x'F_{x'} \tag{1.1}$$

The decomposition is performed on each node in BDDs with respect to the predefined variable. Finally, all duplicate nodes in the preliminary decision tree are merged and all redundant nodes deleted from the tree, resulting in a general graph.

[24] presents the basic strategy of utilizing BDDs to perform equivalence checking. BDD for each primary output bit of compared circuits needs to be built with respect to the corresponding primary inputs. Such established BDDs are then used to check the function equivalence.

However, as already mentioned in [24], the application of traditional BDD representations is limited by large in BDD size. In the worst case, the complexity of building corresponding BDD increases exponentially with the increase of function size. [24] also shows that BDDs can be used in sequential equivalence checking.

When applied to sequential equivalence checking, BDDs represent sets of states instead of representing internal nodes in circuits. After image computation and reachable state computation, one can decide whether two sequential circuits are functionally equivalent or check if given properties are satisfied. Sequential equivalence checking is beyond the scope of this work and will not be elaborated on in this document.

Nevertheless it is known that the application of BDDs in these areas also suffers from size explosion caused by function size and variable order. An example of representing a 2-bit unsigned multiplier with specification $(2a_1+a_0)(2b_1+b_0)$ in BDD is given in Figure 1.5. In this figure p_0, p_1, p_2, p_3 represent primary output bits of the multiplier.

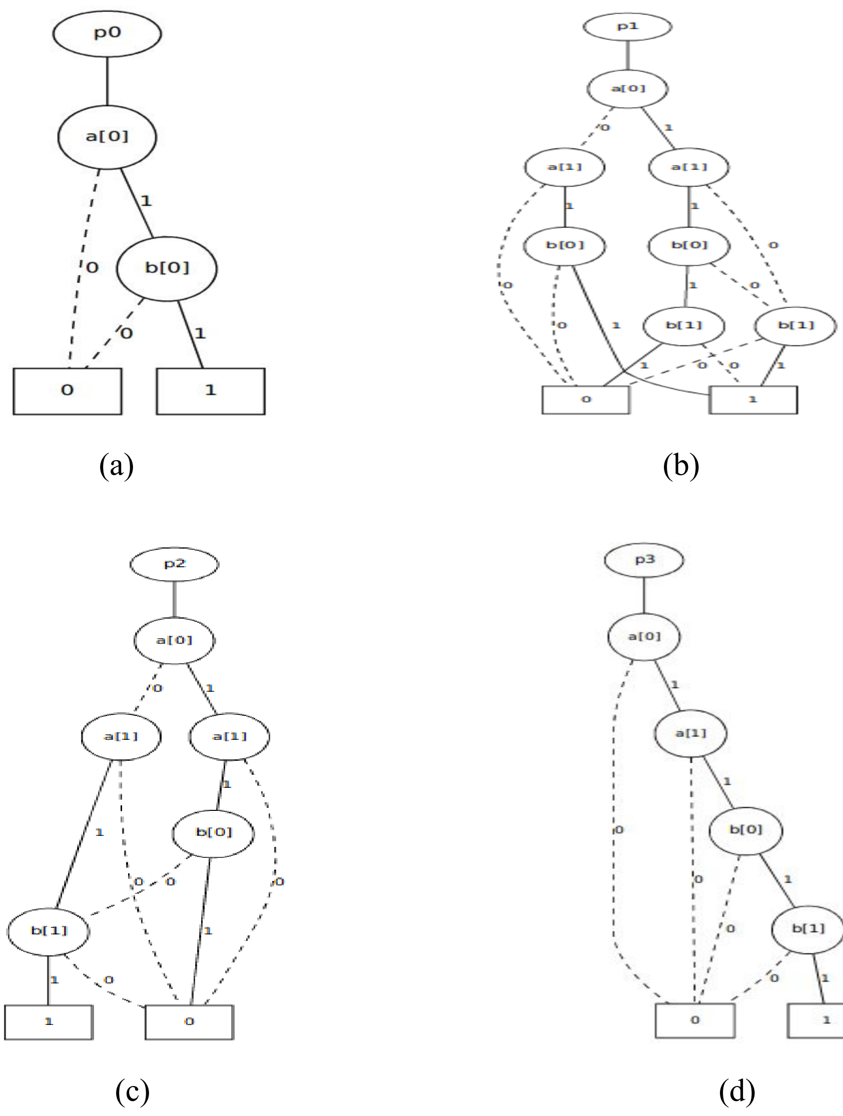


Figure 1.5: An example of BDD for 2-bit unsigned multiplier.

In [27], Burch used BDDs to verify multipliers by representing the multiplier specification in a different way (called as fanout splitting) so as to avoid constructing BDDs of exponential size. Although the BDD size can be reduced to $O(n^3)$, where n is the number of bits of operands in an $n \times n$ multiplier, this method requires construction of specifications that are equivalent to the circuit outputs after fanout splitting. Just as evaluated by [28], requirement of Burch's method is hard to apply to synthesized circuits because the logic may be restructured dramatically. [29] proposed a depth-first algorithm to construct BDDs in order to reduce memory overhead, but with poor spatial locality of reference which degrades the overall performance. [30] focused on building large BDDs in a breadth-first way aiming at improving the efficiency by optimizing memory locality. However, the CPU time overhead increases dramatically with the increase of the size of inputs. Currently the best publicly available BDD manipulation tool is CU Decision Diagram Package (CUDD) developed and maintained by University of Colorado at Boulder [31].

1.3.6 *BMD: An Efficient Representation for Word-Level Functions

Multiplicative Binary Moment Diagrams (*BMDs) [7] enables modeling datapath circuits in word-level data. *BMDs realize efficient representation for important functions that cannot be efficiently represented by BDDs. In *BMDs, edges are associated with weights which can be combined multiplicatively. The differences between BDDs and *BMDs lie in two factors. Firstly, unlike BDDs, *BMDs don't implement point-wise decomposition. In BDDs, each node has two children representing positive factor and negative cofactor of the expression derived by Shannon decomposition respectively. In contrast, *BMDs are based on a rearrangement of

Shannon decomposition, called positive Davio decomposition:

$$F = F_{x'} + x\left(F_x - F_{x'}\right) \tag{1.2}$$

In Equation (1.2) the term $\left(F_x - F_{x'}\right)$ is called the linear moment of f with respect to x. The second difference is that each edge in *BMDs has a specific weight assigned which indicates a multiplicative factor of corresponding node. In BDDs edges just represent the polarity of the decomposing variable (showing if parent node points to a positive cofactor or to a negative cofactor). An example of *BMD representation of function "*8-20z+2y+4yz+12x+24xz+15xy*" is shown in Figure 1.6.

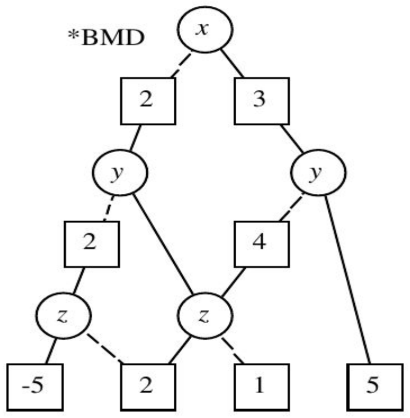


Figure 1.6: An example of *BMD representation of function "*8-20z+2y+4yz+12x+24xz+15xy*"[7].

*BMD shows great advantages in representing designs at the word level. The basic algorithms of applying *BMDs to formal verification area is illustrated are discussed in [7]. The fundamental algorithm uses word-level encoder to encode the bit-level outputs of the circuit while the given specification is also encoded into word level. The *BMD of world-level output expression is compared with *BMD of word-level specification to check whether they are equivalent or not. An improved algorithm, which applies *BMDs to practical design, is called hierarchical verification. In hierarchical verification, the initial design is first partitioned into subcomponents according to its internal word-level structure. Next, the correctness of subcomponents is verified with respect to the corresponding specification. Finally, based on the correctness of subcomponents, the whole design is verified against its specification.

Although *BMDs are found to be much more efficient than BDDs in solving verification problems for large and complex circuits, the applicability of *BMDs is limited by the compulsory requirement of word level representations for the internal structures. This is especially difficult for synthesized or optimized circuits, which often have many irregular structures. Another issue is that, although the *BMD representation can be linear for circuits which have good word level structure, "a mistake in the implementation of integer multiplication logic can cause an exponential explosion of the resulting graph" [32]. *BMDs can represent Boolean expressions with complexity comparable to BDDs. An example of representing 2-bit unsigned multiplier with specification $(2a_1+a_0)(2b_1+b_0)$ using *BMD is given in Figure 1.7.

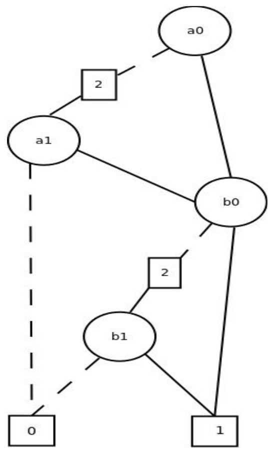


Figure 1.7: An example of BMD for 2-bit unsigned multiplier.

Kronecker multiplicative Binary Moment Diagrams (K*BMDs) [33] as a complement to *BMDs make the representation of Boolean functions easier. K*BMDs incorporate the characteristics of ROBDDs and Edge-Values Binary Decision Diagrams (EVBDDs) [34], and allow dynamic switching between the two. In this way, it is possible to represent both word-level and Boolean-level information in circuits in a single flow. However the deficiency of K*BMDS is that in order to make the diagrams canonical and to make the edges in the diagrams both additive and multiplicative, a set of complex restrictions have to be satisfied.

1.3.7 TED: World-Level Compact Canonical Representation

In [13], a canonical graph-based representation, called Taylor Expansion Diagrams (TEDs), has been proposed that provides efficient verification of designs specified on algorithmic and behavioral levels. The authors of TED noticed that BDD-based verification techniques cannot address the verification problem of larger circuits with a hybrid structure of word-level and bit-level representations satisfactorily (including K*BMDDs mentioned in Section 1.3.6). To address this problem, an entirely different decomposition principle, based on a Taylor series expansion, has been used to decompose the expressions. The circuit is represented as a multi-variate polynomial function. The decomposition is performed on word-level, algebraic variables in the specification, one at a time. For a fixed variable ordering, the resulting TED is canonical.

Given a real, differentiable function $f(x, y, \ldots)$, the result of decomposing $f(x, y, \ldots)$ with respect to variable *x* at an initial point $x_0 = 0$ is:

$$f(x) = f(0) + xf'(0) + \frac{1}{2!}x^2 f''(0) + \frac{1}{3!}x^3 f'''(0) + \ldots \tag{1.3}$$

The *k*-th derivative $f^{(k)}(x=0)$ in equation 1.3 is a *k*-children of variable *x*. Each term in the decomposed equation represents a product of the child-node function and the weight of the edge. Figure 1.8 shows an example of TED result of decomposing expression $(5A+B)(A+2C)$. In Figure 1.8, symbols *A*, *B* and *C* indicate the word-level variables. Each term in the decomposed function is assigned to one of its child node. For example, a dotted line connects the *0*-child with its parent, a single solid line connects the *1*-child with its parent, a solid line labeled "^*i*" connects the *i*-child with its parent, etc. The edges can also be labeled with integers that represents the multiplicative coefficient. For instance, the right most edge in Figure 1.8 has label *(^2 5)*, where *^2* denotes the quadratic child (*2*-child) and constant *5* is the weight of this edge. The reduced TED representation is canonical under fixed variable order.

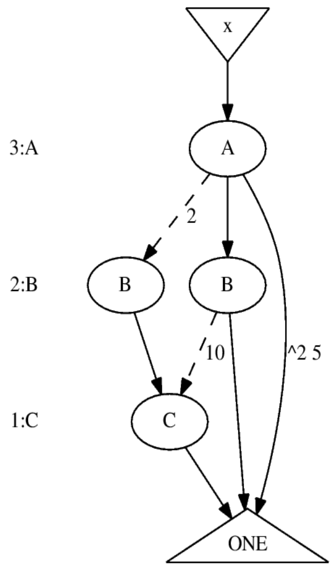


Figure 1.8: An example of TED for *(5A + B)(A + 2C)*.

Moreover, by constraining the integer range to {0, 1} to support Boolean logic in an algebraic way, TED can be modified to represent Boolean functions. TED of a 2-bit unsigned multiplier with specification $(2a_1+a_0)(2b_1+b_0)$ in which all variables are binary is given in Figure 1.9.

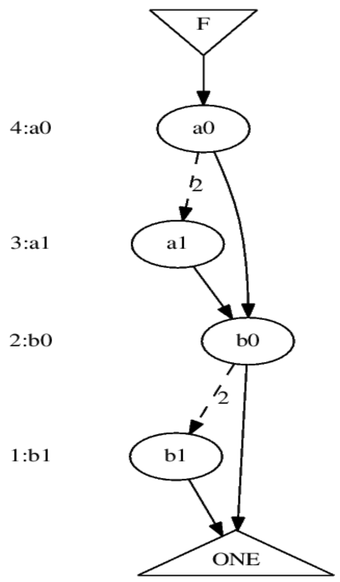


Figure 1.9: TED for a 2-bit unsigned multiplier

The known deficiency of TED is that it cannot represent the function of individual bits in output word with respect to word-level input. Similar to *BMDs, the efficiency of TEDs is affected by the number of variables in the circuit.

1.4 Inspiration for Current Work

The different methods and techniques reviewed above cannot address the verification problem of combinational integer arithmetic circuits efficiently. The proposed work aims at solving this problem efficiently at an algebraic level, treating the function specification (if known) and its implementation as a properly constructed symbolic algebraic system in $\mathbb{Z}_{2^n}$. It derives arithmetic function computed by the circuit from its gate-level implementation, which can be compared with a reference signature to determine whether the circuit is correct. It can also be used as a reverse engineering tool, to learn the function performed by the given circuit. Chapter 2 reviews some advanced methods that try to solve the combinational arithmetic circuit verification problem using symbolic computer algebra. Chapter 3 explains the proposed work, how it differs from previous works and shows the preliminary experiment results.

CHAPTER 2

RELATED WORK

The method proposed in this thesis aims at solving the functional verification problem for combinational arithmetic circuits specified on an arithmetic bit level. In this chapter, several formal verification methods that address the similar application are analyzed.

2.1 Theoretical Background

The underlying mathematical models of formal verification method discussed here are based mainly on symbolic computer algebra [35]. Symbolic computation manipulates expressions with symbolic variables, which are not given any numerical values. In this way, symbolic computer algebra preserves the advantages of formal verification. Conventional formal verification methods, discussed in Chapter 1, typically try to represent primary outputs with respect to primary inputs using certain data structure. However, formal verification methods that will be investigated here generally utilize another interpretation of symbolic computer algebra. The formal verification problem is modeled in this work as a *membership testing* problem between the circuit specification and its implementation as polynomials, based on a computer algebra model. The final goal is to prove that implementation represented by circuit equations satisfies the specification polynomial. This is accomplished by performing a series of divisions of the specification polynomial F by the implementation polynomials $B = \{f_1, ..., f_s\}$. For example, the specification of a multiplier circuit with word-level inputs X, Y and output Z is $F = Z - X \cdot Y$. The implementation polynomials are derived from gate equations, similar to those shown later in Equations 3.1.

To systematically manipulate polynomials, a term ordering ">" is imposed on monomials. The leading term of polynomial g under such ordering is denoted $lt(g)$. Term ordering plays an important role in polynomial reduction used in circuit verification.

Let f, g be polynomials. If a non-zero term t of f_i is divisible by the leading term of g, we say that f reduces to r modulo g denoted:

$$f \xrightarrow{g} r, \text{ where } r = f - \frac{lt(f)}{lt(g)} \cdot g.$$

Similarly, f can be reduced with respect to (divided by) a set of polynomials $B = \{f_1, ..., f_s\}$. This is known as polynomial division modulo B, denoted symbolically as $f \xrightarrow{B}_+ r$ where r is a remainder, with the property that no term in r is divisible by the leading term of any polynomial in B. The sign + refers to the fact that the division process is sequential, using polynomials in B one by one.

Let $B = \{f_1, ..., f_s\}$ be a set of polynomials representing circuit elements (logic gates, adders, arithmetic modules, etc.) and let R be a polynomial ring, $R = F\{x_1, ..., x_n\}$. In fact, in our case R should be defined over integers, $\mathbb{Z}_{2^n}$, rather than a field F. Then, $J = \langle f_1, ..., f_s \rangle$ with $f_i \in \mathbb{Z}[X]$, called an ideal, is a set of all polynomials generated by $\{f_i\}$.

$$J = \langle f_1, ..., f_s \rangle = h_1 f_1 + ... + h_s f_s : h_i \in R \qquad (2.1)$$

The polynomials $f_1, ..., f_s$ are called the bases, or generators, of the ideal J. In our case, each generator is a polynomial model of a circuit module, and the set of generators can be viewed as the implementation of the circuit.

We also need a notion of variety. For a given ideal J, variety $V(J)$ defines a set of all simultaneous solutions to a system of equations $f_1(x_1, \ldots, x_n) = 0$; ..., $f_s(x_1, \ldots, x_n) = 0$. From the circuit perspective, variety contains all the signal values of the circuit produced by any set of primary inputs, over all possible input combinations.

We define a circuit specification as polynomial $F \in \mathbb{Z}_{2^n}[X]$. For example, the specification of a multiplier circuit, $P = A \cdot B$, where A, B are word-level variables, is $F = P - A \cdot B$.

We can now formulate the arithmetic circuit verification problem as follows [36], [37]. Given a circuit represented by the set of generators, $B = \{f_1, \ldots, f_s\}$, and the specification F, the goal is to prove that the implementation (modeled by B) satisfies the specification F. Mathematically, this can be stated that the solution to $F = 0$ agrees with $V(J)$, or, equivalently, that that F vanishes on $V(J)$ [37]. We say that F vanishes on $V(J)$ if F evaluates to 0 for all values of $V(J)$ (which also means the remainder $r = 0$). In computer algebra this problem is known as *ideal membership testing*.

However, if $r \neq 0$, such a conclusion cannot be made; B may not be sufficient to reduce F to 0, and yet the circuit may be correct. To check if F is reducible to zero one must use a canonical set of generators, $G = \{g_1, \ldots, g_t\}$, called Groebner basis, which generates the same ideal as the one based on B, i.e., $J = \langle g_1, \ldots, g_t \rangle = \langle f_1, \ldots, f_s \rangle$. Without Groebner basis one cannot answer the question whether $F \in J$. A number of algorithms have been developed for computing Groebner basis over the field, such as Buchberger

[38], F4 [39], etc., but their computational complexity is prohibitively large for nonlinear arithmetic circuits. Furthermore, these algorithm do not apply directly to rings over integers, $\mathbb{Z}_{2n}$, which is considered in this work.

2.2 Previous Work

Work in arithmetic circuit verification based on computer algebra and algebraic geometry was pioneered by [40] and [41].

In [41] an arithmetic circuit is modeled as a network of arithmetic operators, such as half-adders, comparators, product generators, etc., which in principle can be extracted from the gate-level implementation. These operators are modeled using arithmetic bit-level (ABL) equations, $\{G_j\}$. Authors of [41] (and also [36]) show that for an arbitrary combinational circuit, if the terms of the gate equations $\{G_j\}$ are ordered in the reverse topological order, *{outputs}* > *{inputs}*, then all leading monomials of the polynomials in *B* are relatively prime. As a result, the corresponding set *G* constitutes a Grobner basis, obviating the expensive Grobner basis computation. The verification problem is then formulated as a variety subset problem and solved by reducing the specification modulo *G* to a normal form and testing if it vanishes over $\mathbb{Z}_{2^n}$. Furthermore, in [42], the solution is restricted to binary variables by imposing Boolean constraints, $\langle x^2 - x \rangle$, and solving the problem directly over quotient ring $Z_{2^n}[X]/\langle x^2 - x \rangle : x \in X$. An important simplification comes from the fact that (*Lemma 1*, [42]): “If some polynomial *f* vanishes on $V(J)$ then *f* must be a zero polynomial” and not just a zero function. That is, only the zero polynomial in $Q = Z_{2^n}[X]/\langle x^2 - x \rangle : x \in X$ defines the zero function on $Q^{|x|}$, rendering the zero function test superfluous. This makes it possible to replace the expensive zero function test ($r = 0$?) by checking if r is a zero polynomial. The problem based on this approach was solved using a computer algebra system, Singular [43]. However, this approach is limited to arithmetic bit-level networks composed of half adders and full adders, which need to be extracted from the gate-level implementation. Experimental results show that this is the most expensive part of the process, and such an extraction is not always possible, especially in highly bit-optimized implementations.

In [36], the verification problem is also formulated as ideal membership test but applied to Galois field arithmetic circuits. They have shown that for a special case of Galois Field (GF or F_{2^q}), when the specification F and the ideal J of the circuit constraints (implementation) are in F_{2^q}, then the problem of testing if $F \in I(V(J))$ can be greatly simplified. Specifically, it can be reduced to the ideal membership testing over a larger ideal, $F \in (J + J_0)$, where $J_0 = \langle x^2 - x \rangle$ is an ideal of vanishing polynomials in F_2. Adding J_0 basically restricts variety $V(J)$ to solutions in F_2, i.e., to $V(J) \cap V(J_0)$. It is known from the theory of algebraic geometry [35] that intersection of varieties is equivalent to a union (sum) of ideals.

Similarly to [41], Lv, Kalla, et. al [36] derives term ordering from the topological structure of the circuit, which renders the set of polynomials B (circuit constraints) a Groebner basis, thus obviating the need to perform expensive GB computation. The method uses a customized, F4-style polynomial reduction which is based on a modified Gaussian elimination algorithm [39]. An important feature of this approach is that, by construction, if the remainder $r \neq 0$ then it contains only the primary input variables. Consequently, it can be used to provide a counterexample, or a bug trace, to locate the source of the bug.

However, this method also suffers from some problems that limit its application. For example, this approach applies only to Galois Field networks, and it is not clear if the simplification of the general ideal membership problem to testing for $F \in (J + J_0)$ applies to polynomial rings of integers, $\mathbb{Z}_{2^n}$.

In effect, the two approaches, [42] and [36], managed to reduce the problem to an ideal membership problem, $F \in J$, instead of solving a more complicated problem of checking if $F \in I(V(J))$. Each approach places some limitations on the problem to make it solvable.

Alternative approaches to arithmetic circuit verification were also proposed in [44], [45] and [46]. In [44] an arithmetic bit-level circuit is modeled as a network of half adders, but, in contrast to [42], admits also logic gates. Logic gates are modeled with, or directly derived from, half adders, possibly leaving some of the outputs unused (referred to as floating signals). This model makes it possible to describe an entire network as a system of linear equations. Such a system then represents the implementation of the circuit. The specification is composed of two parts, an input signature, Sig_{in}, a polynomial in primary inputs *(PI)*; and an output signature, Sig_{out}, a polynomial representing the circuit result in terms of the primary outputs *(PO)*. The specification is then defined as the difference between the two signatures, $F_{spec} = Sig_{out} - Sig_{in}$. For example, for a 2-bit adder with inputs a_0, a_1, b_0, b_1 and outputs S_2, S_1, S_0, the specification is defined as $F_{spec} = 4S_2 + 2S_1 + S_0 - (a_0 + 2a_1 + b_0 + 2b_1)$.

The system of linear equations (implementation) is then reduced to a single algebraic expression, called the *circuit signature*, and is compared to F_{spec}. If the signature polynomial is identical to F_{spec}, then the circuit operates correctly according to that specification. If not, the difference between the two, called residual expression, RE, determines a possible mismatch between the implementation and the specification. In [44], a Gaussian-like elimination and standard linear algebra techniques were used to compute the signature, and a canonical polynomial representation TED [47] was used to compare the results.

The shortcoming of this method is that it can only handle linear portion of the network, with linear input signature. Extension to nonlinear circuits is also possible, but it requires additional step to translate the input signature of the linear block into a nonlinear signature in terms of the primary inputs. TDS system [48] based on TED can be used for this purpose. It should be noted that such defined F_{spec} is in fact the same as the specification polynomial F in the works of [36] and [42], and the set of linear equations (or, equivalently, polynomials) forms the basis B of circuit elements (half-adders and logic gates). The resulting RE is then the same as the remainder of the polynomial reduction of F_{spec} modulo set B. The significant difference between these approaches is that in [44] it was not possible to capture the Boolean nature of the signals, i.e., to impose the quotient ring $Z_{2^n}[X]/\langle x^2 - x \rangle : x \in X$ for variables $x_i \in$ X. The authors suggested that Boolean reasoning combined with topological analysis of the circuit can be used to reduce RE to zero, but in the worst case this task could be as difficult as the original problem itself.

In [45], a different model was used, whereby the computation performed by the circuit is viewed as a flow of binary data. For the circuit to be correct, the flow must satisfy a suitably modified Flow Conservation Law. Verification problem was solved by transforming the known input signature into a polynomial in primary outputs only, and checking if the resulting expression matches the output signature (binary encoding at primary outputs). The issue of testing if $RE = 0$ was eliminated by checking the relation between the fanouts and floating signals, that correctly captured the Boolean nature of signal variables. Specifically, the following condition has to be satisfied by the circuit, $\Delta_{fn} - \Sigma_{fl} = 0$, where Δ_{fn} and Σ_{fl} are polynomials representing fanout variables and the floating (unused) signals, respectively. This condition basically states that any additional flow introduced into the network by fanouts, must be compensated by the flow consumed by the floating signals that do not reach primary outputs. In practice, the method is still applicable only to networks with linear input signatures.

In conclusion, the problem of formally verifying integer arithmetic circuits, over $\mathbb{Z}_{2^n}$ remains open. This thesis addresses some stated problems and proposes a robust solution in this domain.

CHAPTER 3

PROPOSED WORK

3.1 Motivation

Original works, [41], [36], in computer-algebra based methods showed that, for combinational circuits with proper (reverse topological) ordering of terms in basis B, the constructed basis over the respective ring (in our case $\mathbb{Z}_{2^n}$) constitutes a Groebner basis. This is true for combinational circuits, which are direct acyclic graphs (DAG). In such circutis the leading monomials are single variables and the leading terms are relatively prime. Whether this fact can help solve the problem of proving equivalence of F over $\mathbb{Z}_{2^n}$ subject to implementation B over $\mathbb{Z}_2$ remains to be proved. In the meantime, we propose to solve the problem bypassing this theoretical issue, and act as follows.

In our case, the specification polynomial $F_{spec} = Sig_{out} - Sig_{in}$ is a ring in $\mathbb{Z}[X]$ with coefficients in $\mathbb{Z}_{2^n}$ and variables in $\mathbb{Z}_2$. In contrast to [45], and work of Kalla et. al, [36] for Galois Fields networks, polynomial F can be nonlinear. This regards the nonlinear circuits, such as multipliers, multiply-accumulator, etc., and any circuit containing logic gates. In this case the input signature and polynomials in B may be nonlinear (see Equations 3.1).

Notice that for polynomials whose terms contain single variables, polynomial division which results in cancellation of terms is equivalent to substitution by the expression corresponding to the substituted variable. This is true in the case of the HA network (ABL model) and in the case of GF networks, composed entirely of XOR and INV gates. Addition of the OR gates complicates the issue, since the polynomial representation of the OR gates $(a + b - a \cdot b)$ contains a nonlinear term, and the same variable appears in more than one term. The same is true for XOR in $\mathbb{Z}_2$, where XOR is represented as $(a + b - 2 \cdot a \cdot b)$ while it does not cause the problem in GF_2, where polynomial for XOR gates are just represented as $a + b$.

All these investigations can be summarized as follows: polynomial division is equivalent to variable substitution and should be done in a reversed topological order, from the gates outputs to the gates inputs. Examples of polynomials used in polynomial division are Equations 3.1 and Equations 3.2. This applies to both linear and nonlinear case. Because of the potential exponential explosion, the division or substitution should be done in the most efficient manner, the topic which will be explored in the remainder of this thesis. The method proposed here extends our work described in [4] from ABL network to gate-level (or hybrid-level) circuit implementation of arbitrary granularity. It offers a robust solution to integer arithmetic verification by computing (extracting) a unique arithmetic function implemented by the circuit, directly from its low-level circuit implementation. From here on, the terms *rewriting*, *substitution* and *unrolling* will be used equivalently.

3.2 Implementation

Our method attempts to solve the functional verification problem of combinational arithmetic circuits at an algebraic level by formulating it as a function abstraction model, i.e., by deriving a unique bit-level polynomial function computed by the circuit directly from its low-level implementation. It uses an efficient, guided elimination technique while trying to avoid the conventional and expensive process of Groebner basis computation and implementing polynomial division.

In our method, the circuit under study is composed of arbitrary elements, such as logic gates and multiple-output arithmetic components. It will be modeled as a network of interconnected bit-level components (modules), each with a finite set of binary inputs and one or more binary outputs. Specifically, a module represents a single-output Boolean logic gate (AND , OR , XOR, INV) or a bit-level arithmetic circuit (half adder, HA , or a full adder, FA) with two binary outputs, carry C and sum S. In this sense, the proposed model admits a hybrid network, composed of an arbitrary collection of logic gates and bit-level arithmetic components. At one extreme, it can be a purely gate-level circuit; at the other, a network composed of arithmetic components only.

Each module m_i in the network is modeled as a polynomial with variables $X = \{x_1, ..., x_n\} \in \mathbb{Z}_2$ (binary) and coefficients in $\mathbb{Z}_{2^n}$ (integers modulo 2^n). More precisely, f_i is a polynomial quotient ring over $Z_{2^n}[X]/\langle x^2 - x\rangle : x \in X$. The restriction to $\langle x^2 - x\rangle$ is dictated by the binary nature of the circuit signals. Sometimes, such a polynomial is referred to as a *pseudo-Boolean* expression, since it represents an algebraic expression, with usual algebraic multiplication and addition operators over Boolean variables. For example, an AND gate $(a \wedge b)$, is expressed by an algebraic equation $p = a \cdot b$, or equivalently by a polynomial $p - a \cdot b$, etc. The following equations summarize algebraic representation of Boolean operators:

$$\neg a = 1 - a$$

$$a \wedge b = a \cdot b$$

Equations 3.1

$$a \vee b = a + b - a \cdot b$$

$$a \oplus b = a + b - 2a \cdot b$$

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

$$HA : 2C + S = a + b$$

Equations 3.2

$$FA : 2C + S = a + b + c_{in}$$

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

We define the verification problem by setting $F_{spec} = Sig_{out}$. We devise a procedure (based on Gaussian elimination combined with term substitution) to rewrite Sig_{out} into Sig_{in} using polynomial representation (shown in Equations 3.1 and 3.2) of the internal circuit elements (gates, adders, etc.). If the resulting Sig_{in} contains only the primary inputs (PI) then it uniquely determines the arithmetic function computed by the circuit. The designer can then determine if the obtained input signature correctly describes the expected function of the circuit by comparing the computed Sig_{in} with given specification. In this procedure, the basic requirement is to perform the substitution in a reversed topological order. The reason for this can be clarified using the following example. In addition, some interesting observations are also found to make the substitution process more efficient.

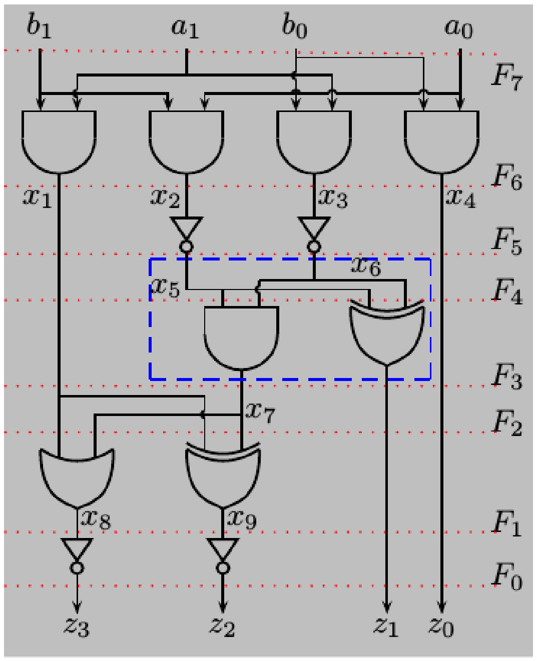


Figure 3.1: An example of gate-level implementation of a 2-bit signed multiplier.

Figure 3.1 shows the gate-level implementation of a 2-bit signed multiplier. This circuit has a hybrid-level structure because because it contains both logic gates and a half adder shown in the dotted box. This circuit is non-linear since its input signature $Sig_{in} = (-2a_1 + a_0)(-2b_1 + b_0) = 4a_1b_1 - 2a_1b_0 - 2a_0b_1 + a_0b_0$ has non-linear terms.

The first step is to construct the equation set for each component in the circuit. The resulting equation set is as follows:

$$z_3 = 1 - x_8 \qquad \text{Equation 3.3}$$

$$z_2 = 1 - x_9 \qquad \text{Equation 3.4}$$

$$z_1 = x_5 + x_6 - 2x_5x_6 \qquad \text{Equation 3.5}$$

$$z_0 = a_0 b_0 \qquad \text{Equation 3.6}$$

$$x_8 = x_1 + x_7 - x_1 x_7 \qquad \text{Equation 3.7}$$

$$x_9 = x_1 + x_7 - 2 x_1 x_7 \qquad \text{Equation 3.8}$$

$$x_7 = x_5 x_6 \qquad \text{Equation 3.9}$$

$$x_5 = 1 - x_2 \qquad \text{Equation 3.10}$$

$$x_6 = 1 - x_3 \qquad \text{Equation 3.11}$$

$$x_1 = a_1 b_1 \qquad \text{Equation 3.12}$$

$$x_2 = a_0 b_1 \qquad \text{Equation 3.13}$$

$$x_3 = a_1 b_0 \qquad \text{Equation 3.14}$$

By definition, the output signature, Sig_{out}, of the circuit is a linear polynomial of the primary output signals. It is uniquely determined by the *n*-bit encoding of the output, provided by the designer. In this example, $Sig_{out} = -8z_3 + 4z_2 + 2z_1 + z_0$. For more general cases, an output signature of any arithmetic circuit with n output bits z_i is represented as follows:

$$Sig_{out} = \sum_{i=0}^{n-1} 2^i z_i$$

By substituting variables in Sig_{out} with their respective expressions, Equation 3.3 to (3.14, we obtain the following sequence of intermediate expressions:

1) By substituting variable z_3 in Sig_{out} using Equation 3.3, Sig_{out} is converted to

$$F_1 = 4z_2 + 2z_1 + z_0 + 8x_8 - 8$$

2) By substituting variable z_2 in F_1 using Equation 3.4, $F_1$is converted to

$$F_2 = 2z_1 + z_0 + 8x_8 - 4x_9 - 4$$

3) By substituting variable x_8 in F_2 using Equation 3.7, F_2 is converted to

$$F_3 = 2z_1 + z_0 + 8x_1 + 8x_7 - 8x_1x_7 - 4x_9 - 4$$

4) By substituting variable x_9 in F_3 using Equation 3.8, F_3 is converted to

$$F_4 = 2z_1 + z_0 + 4x_1 + 4x_7 - 4$$

5) By substituting variable z_1 in F_4 using Equation 3.5, F_4 is converted to

$$F_5 = 2x_5 + 2x_6 - 4x_5x_6 + z_0 + 4x_1 + 4x_7 - 4$$

6) By substituting variable x_7 in F_5 using Equation 3.9, F_5 is converted to

$$F_6 = 2x_5 + 2x_6 + z_0 + 4x_1 - 4$$

7) By substituting variable z_0 in F_6 using Equation 3.6, F_6 is converted to

$$F_7 = 2x_5 + 2x_6 + a_0b_0 + 4x_1 - 4$$

8) By substituting variable x_1 in F_7 using Equation 3.12, F_7 is converted to

$$F_8 = 2x_5 + 2x_6 + a_0b_0 + 4a_1b_1 - 4$$

9) By substituting variable x_5 in F_8 using Equation 3.10, F_8 is converted to

$$F_9 = -2x_2 + 2x_6 + a_0b_0 + 4a_1b_1 - 2$$

10) By substituting variable x_6 in F_9 using Equation 3.11, F_9 is converted to

$$F_{10} = -2x_2 - 2x_3 + a_0b_0 + 4a_1b_1$$

11) By substituting variable x_2 in F_{10} using Equation 3.13, F_{10} is converted to

$$F_{11} = -2a_0b_1 - 2x_3 + a_0b_0 + 4a_1b_1$$

12) By substituting variable x_3 in F_{11} using Equation 3.14, F_{11} is converted to

$$F_{12} = -2a_0b_1 - 2a_1b_0 + a_0b_0 + 4a_1b_1$$

After these12 steps, all equations derived from the circuit have been used. As a result the Sig_{out} has been converted to $F_{12} = -2a_0b_1 - 2a_1b_0 + a_0b_0 + 4a_1b_1$ which matches the given input signature. Furthermore, by analyzing the factored form of Sig_{in}, $Sig_{in} = (-2a_1 + a_0)(-2b_1 + b_0)$, we conclude that this circuit is a 2-bit signed multiplier. Such a factorized form can be obtained using TDS [47] system based on canonical TED representation.

In the substitution procedure shown above, the equation for F_i is in Disjunctive Normal Form (DNF). It can be shown that each term in the DNF equation appears only once, and hence Sig_{out} expression is canonical (This will be formally proved in Chapter 5). Essential of this approach is to improve the efficiency of the substitution process.

First, we must determine which variable to substitute in each step in order to make the cancellation between terms happen as early as possible. This is of great importance for keeping the expression F_i of the transformed Sig_{out} expression in each step as simple as possible in terms of the number of its terms. For example, we will identify variables that depend on common fanouts, as this will increase the number of similar expressions and will increase a chance for simplification and elimination of common subexpressions. For instance, variables in subexpression of F_2, $8x_8 - 4x_9$, depend on common fanout variables x_1 , x_7. As a result, $8x_8 - 4x_9 = 4(2x_8 - x_9)$ after substitution in Equation (3.7) and (3.8) is reduced to $4(x_1 + x_7)$, without introducing a nonlinear term $8x_1x_7$. Hence, expression F_2 can be directly transformed into F_4. If the substitution steps are modified, for example by moving step 4 after step 6, then the Sig_{out} expression after substituting x_9 will be

$$2x_5 + 2x_6 + z_0 + 4x_1 + 4x_5x_6 - 8x_1x_5x_6 - 4x_7 + 8x_1x_7 - 4$$

instead of

$$2x_5 + 2x_6 + z_0 + 4x_1 - 4.$$

In this hypothetical case, we can see that in order to get rid of x_7, Equation (3.9) has to be called again. Hence the size of expression, in terms of product terms, in each step is larger and the number of equations used for substitution is also larger. This is obviously more expensive than the order shown in the original procedure.

Second, we try to simultaneously eliminate all outputs of higher level modules such as adders (if present in the design). Consider, for instance, the dotted box in Figure 3.1, which represents a half adder. As shown by Equation (3.2), the weighted sum of the half adder outputs, $2x_7 + z_1$, can be replaced directly by its inputs, $x_5 + x_6$, thus avoiding unnecessary introduction and elimination of the nonlinear term $4x_5x_6$. As a result, cut F_4 can be directly transformed into F_6 . Such nonlinear terms are particularly harmful if their variables continue to be substituted by other variables, potentially leading to an exponential explosion.

Another important heuristic, which is not shown in example of Figure 3.1 explicitly, is to keep all variables Boolean. We will do this by replacing the expensive division by $\langle x^2 - x \rangle$ (employed by [41] and other symbolic algebra methods) by lowering x^k to x every time variable x is raised to higher degree during substitution. This may happen in cases such as the one shown below.

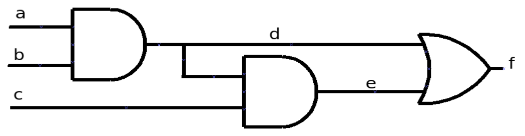


Figure 3.2: An example of keeping variable Boolean.

In Figure 3.2, variable *f* is the output, and variables *a*, *b* and *c* are the binary inputs. If substitutions are performed using strictly algebraic manipulation (multiplication), the expression for *f* will be $ab + abc - a^2b^2c$. However, by maintaining the Boolean value of variables *a*, *b*, *f* will be represented as

$$ab + abc - abc = ab.$$

Other heuristics, noticeably the levelization algorithm, that make the substitution process more efficient are examined in the following chapters. Specifically, these heuristics include:

- Dependency and levelization:

  a) Substitution must follow the reverse-topological order; once a given variable (output of a gate) is substituted by an algebraic expression of the gate inputs, it will be eliminated from the current cut expression and will never be considered again. That is, a variable is substituted for only after substituting all signals in its logical cone. Since the circuit is acyclic, there always exists an ordering of substitutions that satisfies this condition. We refer to this topological constraint informally as "vertical", since it orders variables upwards from primary outputs to primary inputs.

  b) To further increase the efficiency of substitution, another ("horizontal") constraint is imposed on the ordering of the candidate variables at a given transformation step. Specifically, the variables that are at the same logic level (from primary inputs) and have transitive fan-in to common variables should be eliminated together, as this will maximize a chance of the reduction of common terms. It is these variables that define the best cut at each step of the procedure.

- Complex gates:

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

- Binary signals:

  During elimination, the expensive division by the ideal $x^2 - x$, employed by [42], is replaced by lowering x^k to x every time variable x is raised to higher degree during the substitution process. For example, if at any point an expression contains a term xyx, it will be replaced by xy. With this, an expression, such as $xyx-yxy$, will immediately reduce to 0.

- Efficient data structure:

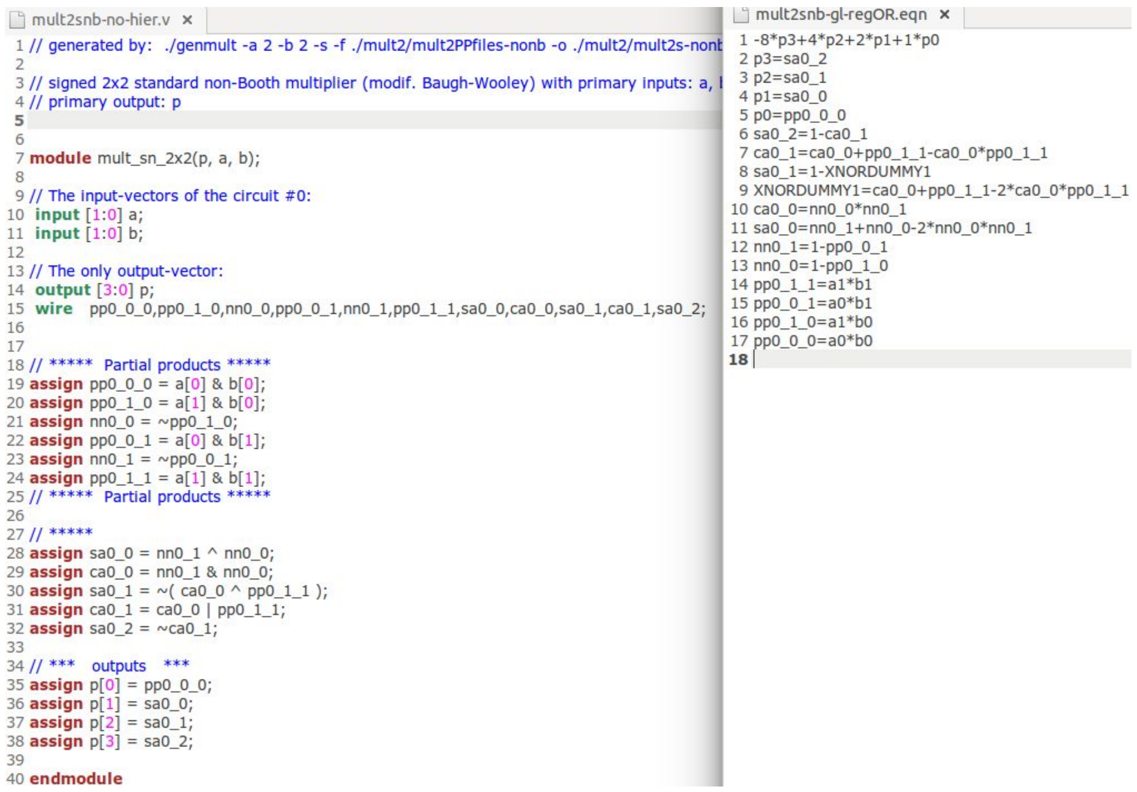
  Our algorithm uses an efficient data structure to support these simplifications and efficiently implement an iterative substitution and elimination process.

CHAPTER 4

IMPLEMENTATIONS AND IMPROVEMENTS OF VARIABLE SUBSTITUTION METHODS

4.1 Preliminary Experiments and Result Analysis

To test the basic variable substitution method presented in Chapter 3, we wrote a prototype program in Python that performs such variable elimination. The basic heuristic applied here is to replace each variable only once and to keep each variable binary. The input file to this prototype program is an equation file which is converted from the Verilog description of the circuit. The format of the equation file is predefined as follows the first line in it must be the given output signature, the subsequent lines are gate equations. This format will be the standard input format for all the following algorithms. The order of gate equations in the equation files is the reverse of the order of logic expressions in the Verilog file. The equations are obtained by translating the original Verilog netlist into a netlist of 2-input *OR*, *XOR, AND* and *INV* gate equations. The output signature provided by user is the linear combination of primary output bits defined by the output encoding. An example of the conversion between the Verilog file and the equation file is shown in Figure 4.1. For example, the first equation on line 19 of the Verilog file is converted to the last equation at line 17 of the equation file. Note that the first line in the equation file is the output signature of the multiplier. For each equation in the equation file, the variable on the left side of the equality sign "=" is the variable that needs to be substituted for in the signature. We refer to this variable as ***target variable***. The polynomial on the right side of the equality sign is the substitution for the ***target variable***. We refer to this polynomial as ***substitution polynomial***.



mult2snb-no-hier.v

```
// generated by:  ./genmult -a 2 -b 2 -s -f ./mult2/mult2PPfiles-nonb -o ./mult2/mult2s-nonb

// signed 2x2 standard non-Booth multiplier (modif. Baugh-Wooley) with primary inputs: a,
// primary output: p


module mult_sn_2x2(p, a, b);

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

// The only output-vector:
 output [3:0] p;
 wire   pp0_0_0,pp0_1_0,nn0_0,pp0_0_1,nn0_1,pp0_1_1,sa0_0,ca0_0,sa0_1,ca0_1,sa0_2;


// *****  Partial products *****
assign pp0_0_0 = a[0] & b[0];
assign pp0_1_0 = a[1] & b[0];
assign nn0_0 = ~pp0_1_0;
assign pp0_0_1 = a[0] & b[1];
assign nn0_1 = ~pp0_0_1;
assign pp0_1_1 = a[1] & b[1];
// *****  Partial products *****

// *****
assign sa0_0 = nn0_1 ^ nn0_0;
assign ca0_0 = nn0_1 & nn0_0;
assign sa0_1 = ~( ca0_0 ^ pp0_1_1 );
assign ca0_1 = ca0_0 | pp0_1_1;
assign sa0_2 = ~ca0_1;

// ***   outputs   ***
assign p[0] = pp0_0_0;
assign p[1] = sa0_0;
assign p[2] = sa0_1;
assign p[3] = sa0_2;

endmodule
```

mult2snb-gl-regOR.eqn

```
-8*p3+4*p2+2*p1+1*p0
p3=sa0_2
p2=sa0_1
p1=sa0_0
p0=pp0_0_0
sa0_2=1-ca0_1
ca0_1=ca0_0+pp0_1_1-ca0_0*pp0_1_1
sa0_1=1-XNORDUMMY1
XNORDUMMY1=ca0_0+pp0_1_1-2*ca0_0*pp0_1_1
ca0_0=nn0_0*nn0_1
sa0_0=nn0_1+nn0_0-2*nn0_0*nn0_1
nn0_1=1-pp0_0_1
nn0_0=1-pp0_1_0
pp0_1_1=a1*b1
pp0_0_1=a0*b1
pp0_1_0=a1*b0
pp0_0_0=a0*b0
```

Figure 4.1: Source Verilog file and converted equation file of a 2-bit signed multiplier

After the equation file is read in, substitutions start from Sig_{out}. The substitution and elimination approach implemented in the prototype program is straightforward. The program repeatedly calls the *clean_substitute* function. This function takes the current intermediate signature and one gate equation encountered as input parameters. After a ***target variable*** is completely substituted, it will return a new simplified intermediate signature to the next call. After all gate equations in the equation file are exhausted, the returned signature will be the computed input signature for the given output signature and gate equations. The pseudo code for the *clean_substitute* function is shown below.

```
Input: one intermediate signature, one gate equation
Output: a new signature after substitution and proper cancellation

// Preparation step:
simplify given signature to DNF ;    // make signature canonical
// Substitution steps:
recognize the target variable targVar to be substituted;
get the polynomial needed to substitute the targVar;
for each literal in signature do
    if literal == targVar then
        substitute targVar by (polynomial);
    end
    continue to next literal;
end
simplify substituted signature and get new signature newSig;
return newSig;
```

Algorithm 1: *clean_substitute* pseudo code.

In Algorithm 1, the *simplify* method is used to perform the term cancellation task by combining terms that have the same monomial. It is very efficiently implemented in the following fashion. In the *simplify* method, each term in the signature is stored in a dictionary with a monomial string as the key and its integer coefficient as corresponding containment. If the monomial of a term already exists in the dictionary, the coefficients are combined. For cases when monomial keys are not stored yet, a new monomial-coefficient pair will be added into the existing dictionary.

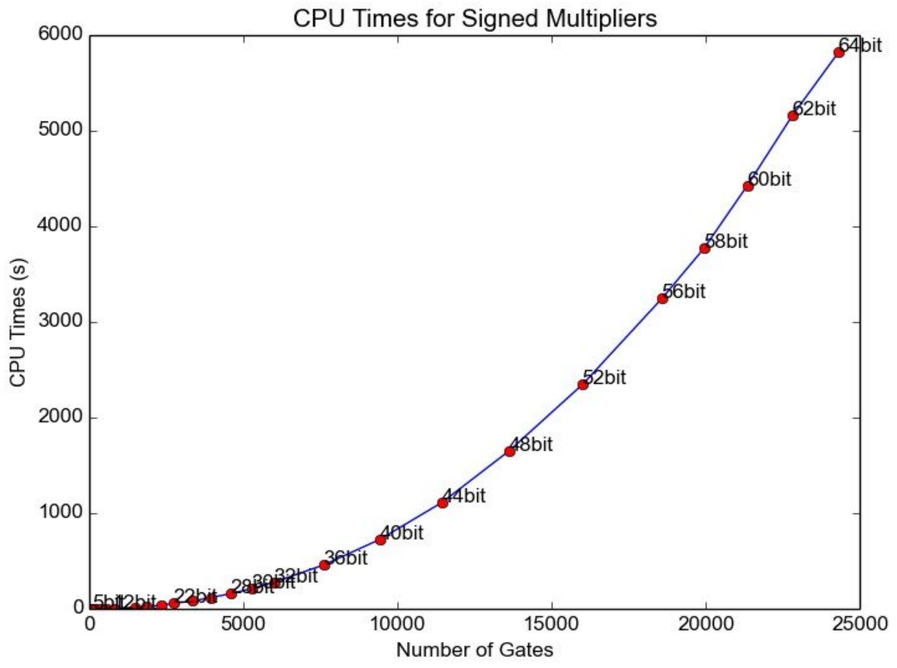


Figure 4.2: CPU times for verifying signed multipliers using *clean_substitute*.

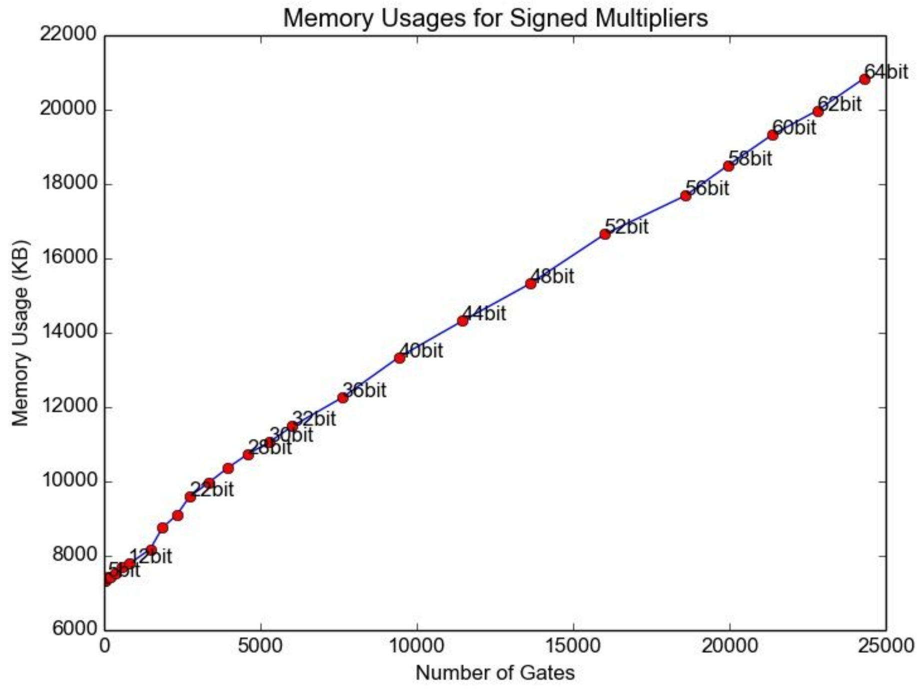


Figure 4.3: Memory usage for verifying signed multipliers using *clean_substitute*.

We performed initial experiments on gate-level arithmetic circuits, such as multipliers. The multipliers were generated by a program Genmult [49]. The tests were run on a PC with an Intel® Core™ i5-3470 CPU @ 3.20GHz × 4 processor, 15.6 GB of memory and a 229.2 GB disk. Our preliminary experiment results are shown in Figure 4.3 and Figure 4.4.

The CPU time results for verifying the signed multipliers, shown in Figure 4.3, demonstrate quadratic dependency on the number of gates. The rudimentary implementation is able to verify signed multipliers up to 64 bits in a reasonable time. Note that the memory usages for our method is very small, less than 15MB for verifying 64-bit signed multiplier, as shown in Figure 4.4, and the main limitation is the CPU time.

Analyzing the preliminary implementation, it is obvious that the *clean_substitute* function consumes most of the CPU time, because it will be called as many times as the number of gates in the circuit. Also it is the most complex function in the program. Since the *simplify* method in the *clean_substitute* function is hard to be further optimized, the chance of improving the efficiency lies in modifying the overall substitution process. As shown in Algorithm 1, the substitution is achieved by comparing each literal in the signature with the ***target variable***. If a match is found, the literal will be replaced by the polynomial that algebraically matches the **target variable**. Once all matches are found, the resulting signature will be simplified and returned to next step.

4.2 Experiments on Improved Algorithm and Result Analysis

One of the heuristics (and the only heuristic implemented up to now) described in

Section 3.2 is to keep variables binary. This means that after simplification, it is not possible to have a term like "$a \cdot a \cdot b$", given that a and b are both binary variables. The term a^k of a binary variable a evaluates always to a itself, so each literal in a term appears exactly once. That is, for each term in the simplified DNF expressions, it is enough to check whether the ***target variable*** is in the expression or not, instead of checking what every single literal is. Algorithm 2 shows the *new_clean_substitute* procedure obtained by modifying the substitution process in *clean_substitute* function is as follows.

```
Input: one intermediate signature, one gate equation
Output: a new signature after substitution and proper cancellation
// Preparation step:
simplify given signature to DNF ;    // make signature canonical
// Substitution steps:
recognize the target variable $targVar$ to be substituted;
get the polynomial needed to substitute the $targVar$;
$term\_list = split\_signature\_to\_terms$;
for each term in term_list do
    $multiplier\_list = term.monomial.split(*)$;
    if $targVar$ in $multiplier\_list$ then
        substitute $targVar$ by (polynomial) in term;
    end
    continue to next term;
end
recombine sustituted terms into new signature $newSig$;
simplify $newSig$;
return $newSig$;
```

Algorithm 2: *new_clean_substitute* pseudo code.

Algorithm 2 differs from Algorithm 1 in the substitution steps. In the *new_clean_substitute* function, each signature is initially split into terms instead of being searched from the first character to the last and each unique term is queried. The substitution is now performed on each ***minterm*** instead of on the whole signature. The irredundancy and uniqueness of each ***minterm*** brings another opportunity for improvement for it allows a much faster substitution method implemented within this algorithm (not shown in Algorithm 2). The *new_clean_substitue* method offers a CPU time complexity that is proportional to the number of terms in the signature. It is significantly better than the performance of the preliminary *clean_substitute* method whose complexity was proportional to the number of characters in the signature. We performed the experiments on the *new_clean_substitute* function using the same benchmark circuits (multipliers generated by Genmult program) and a computer with the same memory and disk spaces.

Figure 4.4 shows the CPU time used for verifying signed multipliers of gate-level implementation. Compared to the data shown in Figure 4.3, the CPU time is only on average one tenth of that in the previous experiments. Comparing Figure 4.5 with Figure 4.4, the improved *new_clean_substitute* function consumes about 1.3 times extra memory space than that consumed by *clean_substitute* function on average. However, considering that the maximum resident memory size used by the *new_clean_substitute* algorithm for verifying a 64-bit signed multiplier is less than 22 Mbyets, such a negligible memory increase is not a issue for today's computers.

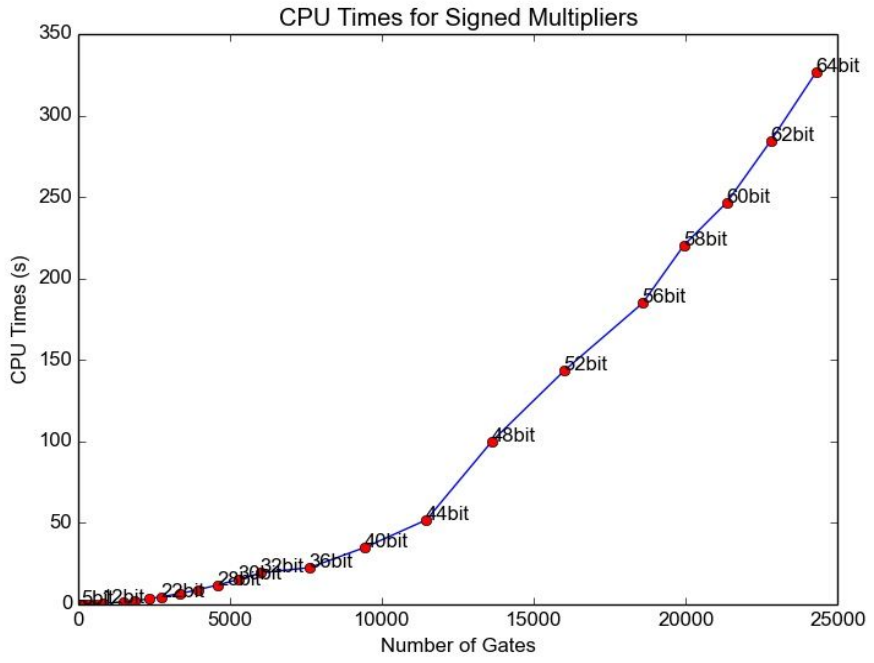


Figure 4.4: CPU times for verifying signed multipliers using *new_clean_substitute*.

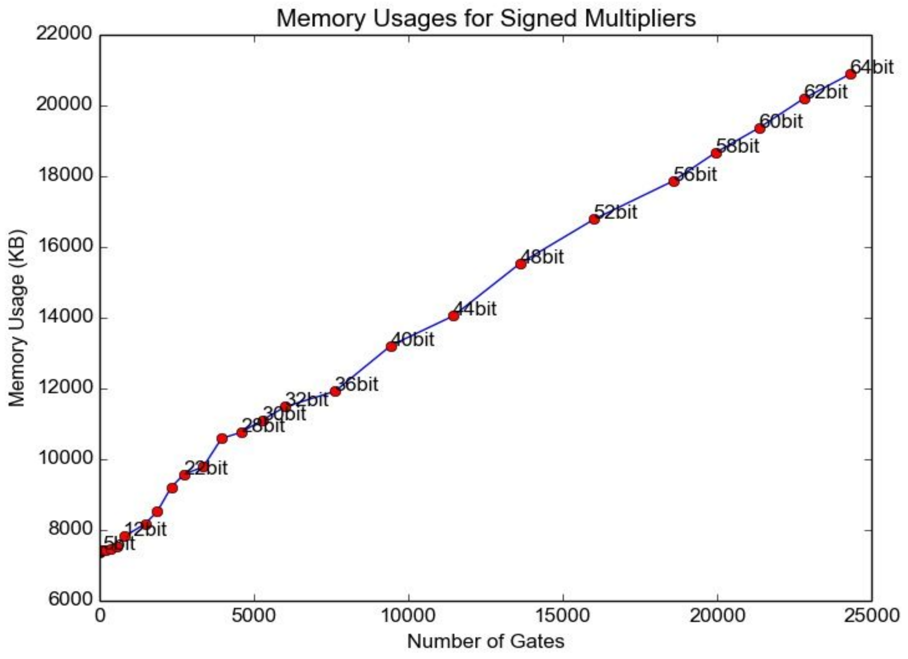


Figure 4.5: Memory usage for verifying signed multipliers using *new_clean_substitute*.

4.3 Experiments on Further Improved Algorithm and Result Analysis

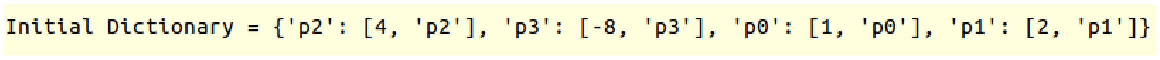
Through investigating Algorithm 1 and Algorithm 2, we know that each time a new logic gate is encountered, the substitution and cancellation method (*clean_substitute* in Algorithm 1 or *new_clean_substitute* in Algorithm 2) will be called.

At the beginning of the *new_clean_substitute* method, the intermediate signature needs to be split into different terms to facilitate variable match. At the end of the *new_clean_substitute* method, to return a new signature in canonical DNF format, the substituted signature (*newSig*) needs to be simplified by combining like terms. As mentioned in Section 4.1, the final simplification method also requires splitting the terms in the intermediate signature and storing them in a dictionary. If the signature needs to be split both at the beginning and the end of the algorithm, why not keep storing the signature in a split way? In this way, one splitting step and one recombination step can be saved every time the *new_clean_substitute* is called. This should improves the CPU time performance.

One thing which needs to be noted is that since we desire to store the signature in a split way, the initial input to the substitution method should assume the same structure. In our case, the dictionary data structure is chosen. The reason of choosing a dictionary data structure is that in Python the *findElement()*, *removeElement()* and *setElement()* methods can all be finished in constant time (which means all operations have time complexity *O(1)* [50]). Remember that the first line of each input equation file is the given output signature; the initial dictionary can be built based on this given output signature. For example, if the circuit is a 2-bit signed multiplier, the dictionary for the output signature

$$-8 \cdot p_3 + 4 \cdot p_2 + 2 \cdot p_1 + 1 \cdot p_0$$

is as follows:



```
Initial Dictionary = {'p2': [4, 'p2'], 'p3': [-8, 'p3'], 'p0': [1, 'p0'], 'p1': [2, 'p1']}
```

Figure 4.6: An example of initial dictionary for a 2-bit signed multiplier.

Figure 4.6 shows the the formation of the dictionary structure used in the further improved algorithm. In this method, the dictionary is an unordered list of (*key (string) : element (list)*) pairs. For each pair, the key is the monomial of a term and the element is a list contains integer coefficient and variable strings in the monomial. For convenience, the integer coefficient is always placed at the first position of the list. For instance, a term "$-3 \cdot a_1 \cdot b_1 \cdot a_{12}$" will be stored in the dictionary in the following way "$\{ \ldots , a_1 \cdot b_1 \cdot a_{12} : [-3, a_1, b_1, a_{12}], \ldots \}$".

When the method begins, in each step, it first checks inside the dictionary if there is any term which contains the ***target variable***. If no such term is found, the method continues to the next substitution step. If terms containing the ***target variable*** are found, substitution will be performed on the terms. During the substitutions new terms will be introduced into the existing dictionary while old terms are removed. For example, assume the ***target variable*** is "a_1", the ***substitution polynomial*** is "$b_1 + c_1 - b_1 \cdot c_1$" (which means the encountered equation is "$a_1 = b_1 + c_1 - b_1 \cdot c_1$", an OR gate) and there is a term "$-2 \cdot a_1 \cdot b_1$" contains the ***target variable,*** "a_1". Term "$-2 \cdot a1 \cdot b_1$" will be transformed to "$-2 \cdot b_1 - 2 \cdot c_1 + 2 \cdot b_1 \cdot c_1$". The substitution process will add new terms "$-2 \cdot b_1$", "$-2 \cdot c_1$" and "$- 2 \cdot b_1 \cdot c_1$" into the current dictionary while removing term "$-2 \cdot a_1 \cdot b_1$" from the dictionary.

```
Input: a term dictionary $termDict$, one gate equation
Output: an updated term dictionary $termDict$
// Substitution steps:
recognize the target variable $targVar$ to be substituted;
get the $polynomial$ needed to substitute the $targVar$;
for $each\ term\_key$ in $termDict$ do
    if $targVar$ in $term\_key$ then
        substitute $targVar$ in $term\_key$ by ($polynomial$);
        count $newTerms$ in new simplified polynomial;
        remove $term\_key$ from $termDict$;
        for $each\ newTerm$ in $newTerms$ do
            if $newTerm$ exists in $termDict$ then
                update the $termDict$;
            else
                add $newTerm$ into $termDict$;
            end
        end
    end
    continue to next $term$;
end
return $termDict$;
```

Algorithm 3: *better_clean_substitute* pseudo code.

Algorithm 3 shows the pseudo code of the new algorithm. We refer to it as the *better_clean_substitute* algorithm. The *better_clean_substitute* algorithm treats the changing signature as a term dictionary modified on the fly. Compared with the previous versions, the main advantage of the *better_clean_substitute* algorithm comes from the uniform dictionary data structure all throughout the process. After all the equations are exhausted in the substitution step, a final dictionary will be returned. The last task, which is trivial, is to concatenate all the pairs in the final dictionary into a signature polynomial. That is the computed input signature which will be compared with the expected input signature. The computed input signature basically determines the function of the circuit and is unique for a given output signature (see Chapter 5).

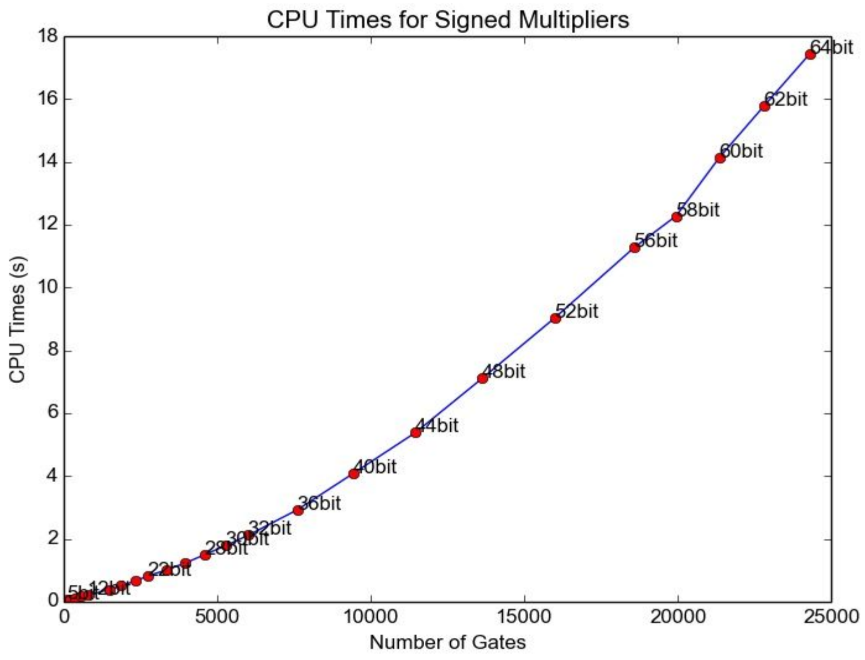


Figure 4.7: CPU times for verifying signed multipliers using *better_clean_substitute*.

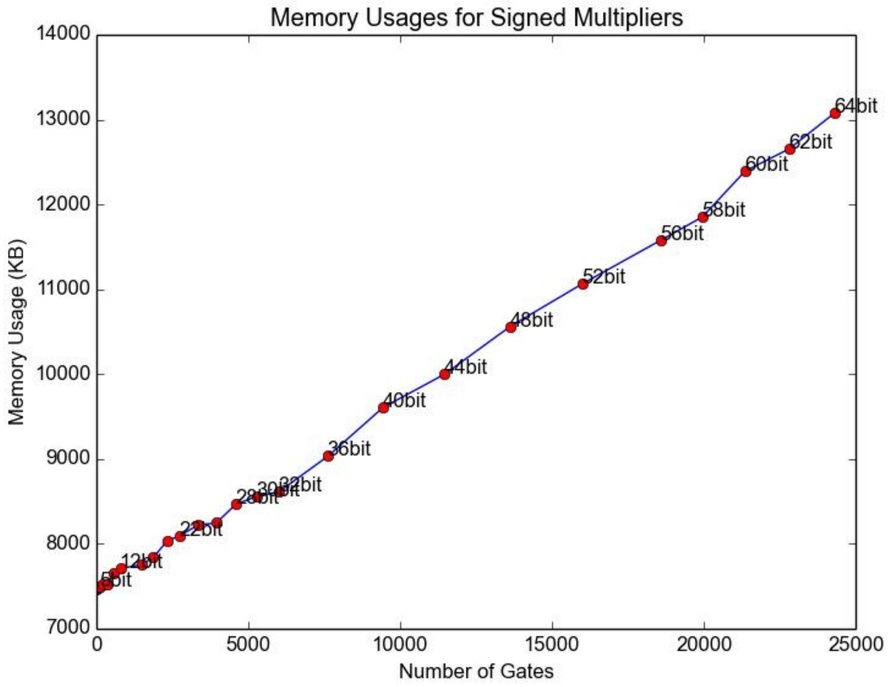


Figure 4.8: Memory usage for verifying signed multipliers using *better_clean_substitute*.

The set of experiments were performed using the *better_clean_substitute* method on the same benchmark circuits, a set of standard array based signed non-booth multipliers ranging from 2 bit to 64 bits. The hardware configuration is the same, a PC with an Intel® Core™ i5-3470 CPU @ 3.20GHz × 4 processor, 15.6 GB memory and a 229.2 GB disk. The experimental results are shown in Figure 4.7 and Figure. 4.11. Figure 4.7 shows dramatic improvement in CPU time performance. Comparing it with the result shown in Figure 4.4, the *better_clean_substitute* method on average displays nearly a *20* times improvement over the *new_clean_substitute* method explained in Section 4.2. The time complexity of the *better_clean_substitute* method remains at *O(n),* while *n* is the number of gates in circuit, but with a much gentler slope. The memory usage shown in Figure 4.8 is nearly 50 percent lower than that of the *new_clean_substitute* method (~13,000 KB versus ~21,000 KB memory used for verifying a 64-bit multiplier).

Overall, the modification of the data storing structure (from string to dictionary) greatly improves the efficiency with respect to both CPU time and memory usage. But the searching algorithm itself has not been optimized that much. In both the *new_clean_substitute* method and the *better_clean_substitute* method, to locate a ***target variable***, the ***target variable*** has to be compared with every single literal in every term of the intermediate signature (Even though in *better_clean_substitution* we check the existence of the ***target variable*** using Python's "***in***" operator instead of comparing it with each literal in one term, the basic thought is the same). Another performance improvement chance lies in optimizing the searching algorithm for ***target variable***. The following section explains the concrete optimization and performance improvements.

4.4 Experiments with Improved Algorithm and Result Analysis

It is well known that the outstanding advantage of using the dictionary data structure is the constant referring time. In previous algorithms, we enjoyed the dictionary's power in implementing the simplification method. The *simplify* method is used directly as a function, or indirectly as part of a function, in the first three algorithms. This can be illustrated with the following example in the *better_clean_substitution* method. Assume there is a term "$2a_1a_2$" in the intermediate signature. After substitution using the equation "$a_1 = b_1 + c_1$", we got two new terms "$2a_2b_1$" and "$2a_2c_1$". In order to update the term dictionary (*termDict* in Algorithm 3), we first check if the key "a_2b_1" exists in the current dictionary. If the key "a_2b_1" is in the dictionary, then the list element indexed by the key will be returned. Suppose the term key has a paired list element such as "$[-4, a_2, b_1]$", the first element "-4" in the list is an integer which represents the coefficient of monomial "a_2b_1" before the substitution. Then the coefficient will be updated to, 2 because $-4 + 2 = 2$, and the list indexed by key "a_2b_1" is updated to "$[-2, a_2, b_1]$". If "a_2b_1" is not one of the keys of the dictionary, then a new element "$a_2b_1 : [2, a_2 , b_1]$" will be inserted into the dictionary. The other term "$2a_2c_1$" can be added to the dictionary similarly. This method outperforms the common character by character string search method in that it can immediately find the term needs to be updated in the dictionary. We noticed that the same method can be applied to locate all terms that contain the ***target variable*** in a intermediate signature in constant time. The detailed method is as follows.

```
Input: a term dictionary $termDict$, one gate equation
Output: an updated term dictionary $termDict$
// Substitution steps:
1  recognize the target variable $targVar$ to be substituted;
2  get the $polynomial$ needed to substitute the $targVar$;
3  if $targVar$ in $termDict$ then
4      $targDict = termDict[targVar]$;
       // delete old terms and get new terms:
5      delete $termDict[targVar]$ from $termDict$;
6      for each $target$ in $targDict$ do
7          delete $target$ from related sub-dictionaries in $termDict$;
8          substitute $targVar$ in $target$ by ($polynomial$);
9          store terms in substituted expressions into $tempDict$;
10     end
       // update $termDict$:
11     for each $newTerm$ in $tempDict$ do
12         if $newTerm$ exists in $termDict$ then
13             update the $termDict$;
14         else
15             add $newTerm$ into $termDict$;
16         end
17     end
18 end
19 return $termDict$;
```

Algorithm 4: *dict_substitute* pseudo code.

As both the substitution step and the *simplify* method rely heavily on dictionary data structure, we refer to the algorithm proposed in this section as *dict_substitute.* Each substitution process can be divided into three subsections. In the first subsection, terms that are going to be substituted are deleted from the main dictionary (*termDict* in this case) and stored in a temporary dictionary. In the second subsection, the terms deleted in the last step are substituted with respect to the ***target variable***. The final step updates the main dictionary based on the substitution result in step 2. The advantage of the *dict_substitution* method is that it can locate all the terms that are going to be substituted in each step in constant time *O(1)*, while previously the terms were found by checking the availability of the ***target variable*** in each term in the main dictionary. This advantage is achieved by using a subtler dictionary structure, compared with the one in the *better_clean_substitute* algorithm. The following example gives a more intuitive impression of the advantage of the *dict_substitution* method.

Example: Assume that the initial signature that starts off the substitution process is "$2a_1a_2 + 2a_1b_2 + 2b_2c_1$" which comprises three different terms "$2a_1a_2$", "$2a_1b_2$" and "$2b_2c_1$". In contrast to the dictionary structure in the *better_clean_substitute* algorithm, which stores "*monomial : list*" pairs, the *dict_substitute* stores "*variable : sub-dictionay*" pairs into the main dictionary. For instance, the three terms in this example are stored as:

$$\{a_1 : \{a_1a_2 : 2,\ a_1b_2 : 2\},\ \ a_2 : \{a_1a_2 : 2\},\ \ b_2 : \{b_2c_1 : 2,\ a_1b_2 : 2\},\ \ c_1 : \{b_2c_1 : 2\}\}$$

In this dictionary, keys are distinct variables that appeared in the current signature. The sub-dictionary indexed by one variable key stores all the terms that contains the key variable. In the sub-dictionary, each related term is stored as a "*monomial : coefficient*" pair. Also it is worth noticing that some terms are stored more than once in the main dictionary. For example, the terms "$2a_1a_2$", "$2a_1b_2$" and "$2b_2c_1$" are stored twice each. Basically, one term has as many copies in the main dictionary as the number of variables in the term, because the main dictionary is a variable-indexed dictionary.

Assume that one equation encountered during the substitution process is "$c_1 = a_1 + a_2$" (it is not necessarily the gate model that we have used, but is simple enough to explain the idea). Given the equation, the ***target variable*** in this case is "c_1" and the polynomial used to substitute "c_1" is "$(a_1 + a_2)$". In the first step, we use the ***target variable*** to locate all terms that are going to be substituted, put them into a temporary dictionary (*targDict* in Figure. 4.11) and delete them from the main dictionary, *termDict*. In this step, it is important to thoroughly delete all the copies of the terms that we want to get rid of from the main dictionary. In this example, the main dictionary *termDict* after deletion is:

$$\{a_1 : \{a_1a_2 : 2,\ a_1b_2 : 2\},\ \ a_2 : \{a_1a_2 : 2\},\ \ b_2 : \{a_1b_2 : 2\}\}$$

Two copies of term "$2b_2c_1$" are both deleted from sub-dictionaries indexed by "c_1" and "b_2" respectively. In the meantime, the dictionary *targDict* which stores target terms that are going to be substituted has the following format:

$$\{c_1 : \{c_1b_2 : 2\}\}$$

In the second step, the substitution is performed on term "c_1b_2" using polynomial "$a_1 + a_2$". After substitution, we have two new terms, "a_1b_2" and "a_2b_2" each with the coefficient "*2*". In this step, new terms achieved by substituting some old terms in the main dictionary will be combined if they have the same monomial. In this example, as the monomials of the new terms are different, they will be directly put into a temporary dictionary, *tempDict,* as follows.

$$\{a_2b_2 : 2,\ a_1b_2 : 2\}$$

Notice that the structure of *tempDict* is different from *termDict.* In *tempDict* we use "*monomial : coefficient*" pairs because they are easier to manipulate.

In the last step, we insert the new terms achieved in the second step into *termDict*. One thing we need to be aware of is that a new term shall be added into multiple sub-dictionaries which are indexed by variables in that new term, respectively. For example, in this case, monomial "a_1b_2 " is already in the *termDict*, so we just need to update its coefficient to *4*, which equals its original coefficient, *2*, plus the coefficient, *2*, of the newly added term. Another monomial "a_2b_2" is not in the *termDict*, so we need to insert it into two sub-dictionaries which are indexed by "a_2" and "b_2" separately. The updated *termDict* looks like this:

$$\{a_1 : \{a_1a_2 : 2,\ a_1b_2 : 4\},\ \ a_2 : \{a_1a_2 : 2,\ a_2b_2 : 2\},\ \ b_2 : \{a_1b_2 : 4,\ a_2b_2 : 2\}\}$$

The above three steps explain the basic idea in the *dict_substitute* algorithm. Besides this there are many other internal function improvements, like a function to recognize primary output variables, a function to implement fast substitution, a function to determine coefficient and monomial of a term, etc. These functions are not discussed here. Readers can check them in the Appendix if interested.

The experiments performed on the *dict_substitute* algorithm utilize the same set of hardware configurations and benchmark circuits as the previous tests use. The results are shown in Figure 4.9 and Figure 4.10. Figure 4.9 shows the linear complexity of CPU time with respect to the numbers of gates in the circuits. This should be compared with the results with Figure 4.7, *dict_substitute* method offers a CPU time complexity improvement of 8x over *better_clean_substitute* method at an extra memory cost (less than *25%*).

Figure 4.11 and Figure 4.12 show the comparisons of CPU time performance and memory usage of four different substitution algorithms. After applying the optimizations of data structure and algorithm, our current best algorithm *dict_substitute* has achieved a 3000x improvement in CPU time and even better memory usage compared with the preliminary *clean_substitute* algorithm.

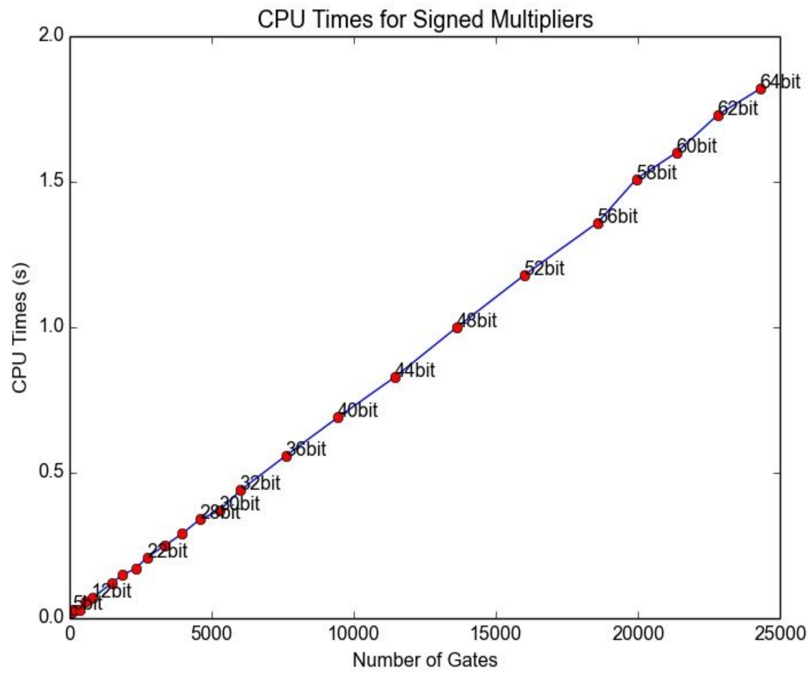


Figure 4.9: CPU times for verifying signed multipliers using *dict_substitute* method.

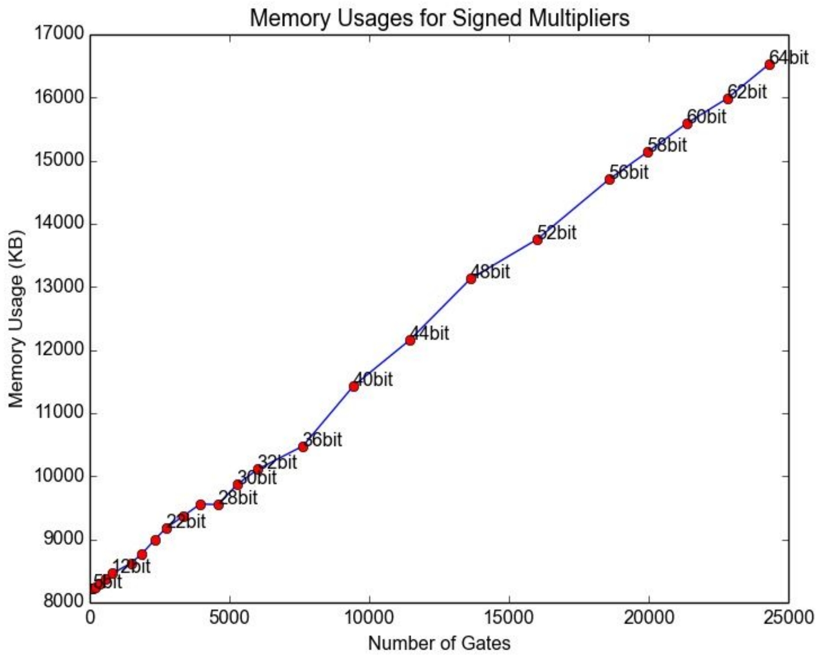


Figure 4.10: Mem usage for verifying signed multipliers using *dict_substitute* method.

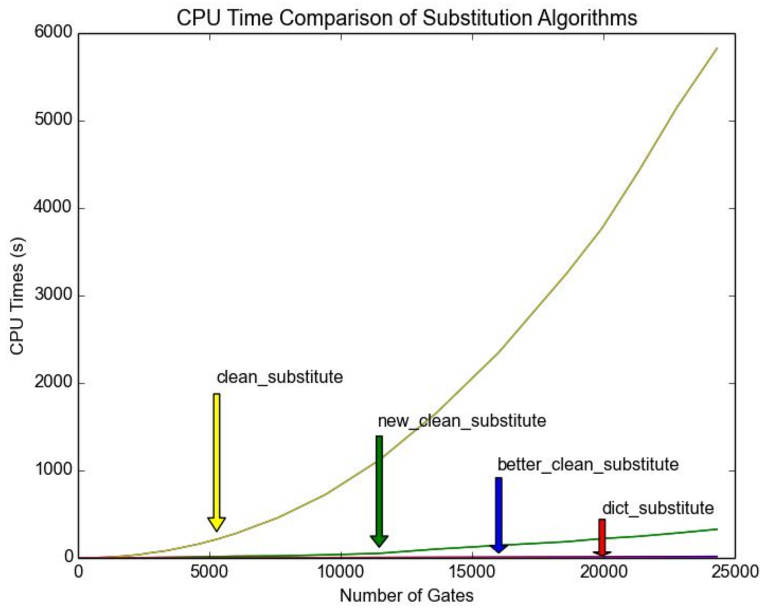


Figure 4.11: CPU time comparison of four substitution algorithms.

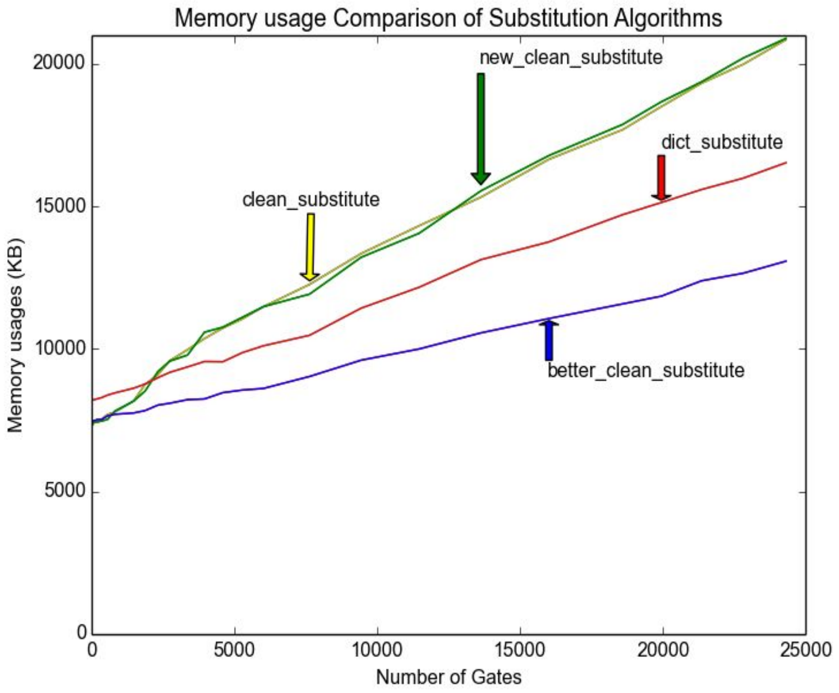


Figure 4.12: Memory usage comparison of four substitution algorithms.

In Figure 4.12, the *better_clean_substitute* method has the best memory usage performance because it neither needs to store intermediately computed signatures nor needs it to store redundant copies of terms that *dict_substitute* algorithm does. The *clean_substitute* algorithm and the *new_clean_substitute* algorithm have similar memory usage characteristics, because in both algorithms an intermediate signature needs to be built after each substitution step. Although they consume the largest memory space, the idea is still useful. If we consider the scenario that there is a bug in the circuit, we may need to know what intermediate signatures look like to determine the location and cause of the bug. In this situation, it is necessary to spend much more time and memory to monitor the verification process in order to analyze circuits. Actually, in the final version of the verification software, this function is also included as an alternative option to show the elaborated intermediate verification information.

CHAPTER 5

THEOREM

In this chapter we prove that the variable substitution method is theoretically correct.

Essential part of the described approach is the following theoretical result about the correctness and uniqueness of the computed input signature. Here, "correct" means that the result is the same as if it were computed with Boolean methods. This result applies to combinational circuits, but it can be readily extended to sequential circuits by unrolling the circuit over a fixed number of time frames into a combinational circuit (bounded model).

Theorem: Given a combinational circuit composed of basic logic gates, described by polynomial Equations (3.1), input signature Sigin computed by the proposed procedure is unique and correctly represents the arithmetic function implemented by the circuit.

Proof: The proof of correctness hinges on the fact that each internal signal is correctly represented by an algebraic expression, i.e., such an expression evaluates to a correct Boolean value. Specifically, it can be easily verified that Equations (3.1) are the correct algebraic representations of basic Boolean functions. Hence, any logic function that is expressed recursively by Equations (3.1) must evaluate to a correct Boolean value; once the polynomial is reduced by removing redundant terms, the algebraic representation is unique. Example: *XOR* function, $f = a \oplus b = a'b + ab'$, can be written as an algebraic function $f = (1 - a)b + a(1 - b) - ((1 - a)b)(a(1 - b))$, which reduces to a unique form, $a + b - 2ab$. Hence, a PO signal is correctly represented by variables in its logic cone, up to the primary inputs. Therefore, Sig_{out}, which is the weighted sum of the output signals, is eventually replaced by Sig_{in}. For this reason such computed Sig_{in} is a correct algebraic representation of the circuit.

The proof of uniqueness is based on induction on i, the step when polynomial F_i is transformed into F_{i+1}. Base phase: polynomial $F_0 = Sig_{out}$, a linear combination of primary outputs, is unique. Also, as discussed above, algebraic representation of each logic gate is unique. Induction step: Assuming that F_i is unique, we must prove that F_{i+1} is unique. Recall that each variable in F_i represents output of some logic gate; during the transformation process it is substituted by a unique polynomial of that gate. Since the circuit is combinational (it has no loops) and the substitution is done in reversed topological order, at each step i a variable in F_i is replaced by a unique polynomial in new variables. Hence, polynomial F_{i+1} derived from F_i by such substitution is also unique.

CHAPTER 6

LEVELIZATION ALGORITHMS

The heuristics proposed here attempt to accelerate the verification method by exploring the circuit structure. The method is based on the variable substitution method described in Chapter 4 and will utilize the heuristics mentioned in Chapter 3. The goal is to make the cancellation between terms happen as early as possible by selecting the most proper ***target variable*** to be substituted in each step. The proposed levelization algorithm is the base algorithm, and additional heuristics progressively enhance the algorithm's performance.

Initial approach considered building an equation pool with the gate equations in arbitrary order. The equation pool will be passed through the variable substitution process iteratively until the signature is comprised of only primary inputs. In this process, one gate equation in the pool may need to be used multiple times. This method is time consuming if the order of processing equations is not strictly topologically reversed. As a result, gates that are topologically far away in the circuit may be substituted in the same iteration. This behavior decreases the opportunity for cancellation and increases the number of substitution rounds. Taking these factors into consideration, we try to first specify the order of the equations derived from the circuit prior to passing them to the substitution step as the inputs. As mentioned in Chapter 3, we expect to derive a well-defined order of the gate equations to utilize each gate equation just once in the entire substitution process.

6.1 Breadth-First Search (BFS) Levelization Algorithm

The first method that we implemented tries to levelize the gate equations in a BFS fashion. The algorithm is shown in Algorithm 5.

```
Input: dictionary eqn_rela_dict stores input-output relationships of
       each gate, dictionary level_dict stores levelized gates, list
       targets stores variables to be levelized
Output: updated term dictionary level_dict
level number cnt = 0;
while list targets is not empty do
    list parents = emty list;
    for each target in targets do
        assign level number to target;
        store target and its level number to level_dict;
        find target's parents, store them in list parents;
    end
    targets = parents;
    cnt++;
end
return level_dict;
```

Algorithm 5: *BFS_Levelization* pseudo code.

Initially, in the *BFS levelization* Algorithm, the variable list ***targets*** just stores one primary output bit. That is the current levelization process will start from this primary output bit. The process levelizes all gates that belong to the logic cone of the primary output bit. We repeat the levelization process on all of the primary output bits to completely levelize all gates in the circuit.

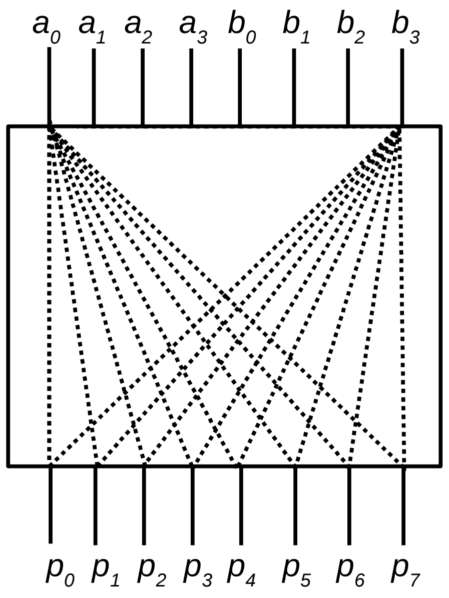

a_0 a_1 a_2 a_3 b_0 b_1 b_2 b_3
p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7

Figure 6.1: *BFS_Levelization* methodology.

Figure 6.1 illustrates the levelization process for a 4-bit multiplier circuit. The eight primary output bits are p_0 to p_7. The levelization algorithm runs eight times in total. Note that, in general, there is an overlap between any two logic cones in the figure. These areas represent gates that participate in generating multiple primary output bits. A simple example which illustrates such logic cones specifically is shown in Figure 6.2.

Assume that first levelization iteration starts at p_0. The *INV* gate *g0*, which generates p_0, is levelized into level *0*. The *AND* gate *g1*, which generates the input signal for *g0*, is assigned to level *1*. The parents of gate *g1* are *AND* gate *g2* and *OR* gate *g3*, so they are put in level *2*. This method is referred to as a BFS algorithm, because all parents of gates in the current level will be found and put in the next higher level.

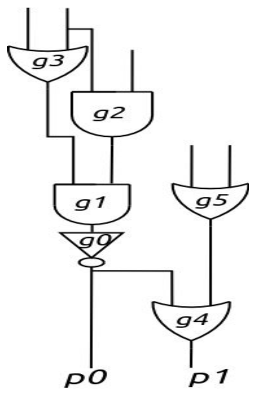


Figure 6.2: Level number relaxation example.

The second iteration starts from primary output bit p_1. The *OR* gate *g4*, which generates p_1, is assigned to level *0*. As the figure shows, gates *g0*, *g1*, *g2* and *g3* are in the overlapping area of logic cones, because all of them participate in generating both p_0 and p_1. For gates in such cones, a relaxation method is used to update their level numbers. Specifically, if a gate to be levelized is already in the dictionary (*level_dict* in Algorithm 5), the relaxation method chooses the larger one between its old level number and the new number intended to assign. For example, in Figure 5.2, the *INV* gate *g0* was assigned to level *0* in the first levelization iteration. However, it will be reassigned to level *1* after the next levelization process starts from p_1, because *g0* generates one of the parents of *g4*, and *g4* is in level *0*. The same applies to other gates in the overlapping area. The final levelization result is

Level 0 *: g4;* ***Level 1*** *: g0, g5;* ***Level 2*** *: g2;* ***Level 3*** *: g2, g3;*

After completely levelizing all gates in the circuit, the equations in the levelized order are used in the substitution method as inputs. The substitution then starts from the gates in the lowest level to the gates in the highest level.

We performed a set of experiments on the signed multipliers used in the previous experiments. We first use the *BFS levelization* algorithm to levelize these circuits. After that we use the variable substitution program discussed in Section 4.4 to verify the levelized circuits. Table 6.1 shows the CPU time performance comparison of verifying circuits before and after levelization. It stops comparison on a 40-bit multiplier because the levelization for the 40-bit multiplier is too time consuming (over one hour).

Signed multipliers	Number of gates	Levelization time (s)	Verification time before levelization (s)	Verification time after levelization (s)	Verification mem after levelization (KB)
4-bit	80	0.01	0.02	0.01	8220
8-bit	352	0.08	0.03	0.04	8288
12-bit	816	0.81	0.07	0.08	8428
16-bit	1472	5.93	0.12	0.12	8572
20-bit	2320	28.67	0.17	0.19	8848
24-bit	3360	106.07	0.25	0.28	9212
28-bit	4592	319.96	0.34	0.37	9516
32-bit	6016	826.38	0.44	0.48	9812
36-bit	7632	1926.31	0.56	0.62	10232
40-bit	9441	4082.60	0.69	0.76	11032

Table 6.1: CPU time comparison of circuits before and after levelization.

Seemingly, the table shows that the *BFS levelization* algorithm does not work as expected. Firstly the levelization process itself is time consuming, secondly the levelized circuits need more time to be verified relative to circuits before levelization.

The first problem is caused by the implementation of the BFS algorithm. Through profiling the execution process of the algorithm, we discovered that most of the levelization time is spent in simply adding and removing elements into and out of lists, whereas, updating the level dictionary does not consume much time. The second problem occurs because levelization breaks the circuit structure of multipliers generated by Genmult. The internal structure of those multipliers is based on Wallace tree. The well-designed original circuits generated by Genmult have very organized structure, a connected netlist of half adders. The substitution process runs in units of half adders when verifying the original circuits. This provides cancellations in consecutive equation substitutions which make the process very efficient. However, the levelization process often breaks this regularity in circuits, thus increasing the verification time.

Nonetheless, the levelization algorithm is very useful to verify synthesized circuits. In the following experiments, we tried to verify circuits synthesized by ABC *[51]*.

1) We first use ABC to synthesize the unsigned CSA multiplier generated by BenGen [52]. The resulting circuit is referred to as the *synthesized circuit*.
2) As ABC "strashes" the circuit to *AIG* structure during synthesis, we need to map the *synthesized circuit* into other kinds of gates. In our experiments, we do the mapping using a standard library, derived from the *mcmc.genlib*. The derived library contains 2-input *AND*, *OR*, *XOR*, *NOT*, *NOR*, *XNOR* gates and 3-input *OAI*, *AOI* gates. The library that we used is shown in Figure 6.3. We refer to this circuit as *mapped circuit*.

Strash: Structural hash, an optimization procedure used by ABC to detect and combine nodes of the same functionality.

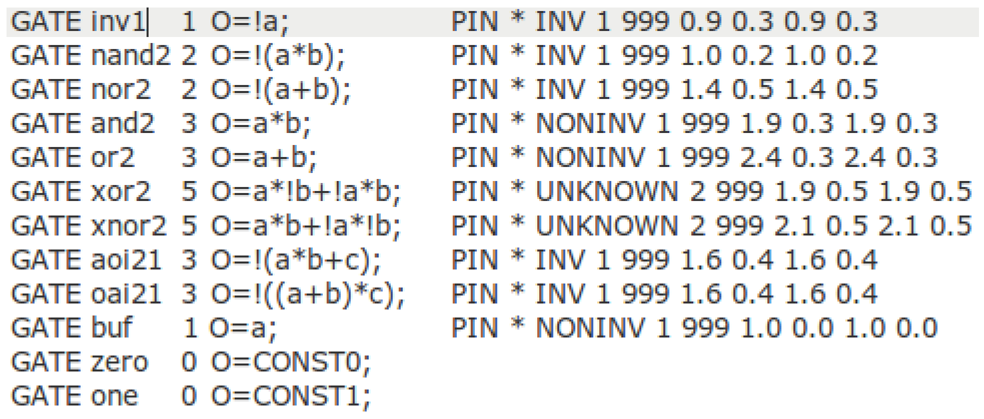

```
GATE inv1   1 O=!a;          PIN * INV 1 999 0.9 0.3 0.9 0.3
GATE nand2 2 O=!(a*b);       PIN * INV 1 999 1.0 0.2 1.0 0.2
GATE nor2  2 O=!(a+b);       PIN * INV 1 999 1.4 0.5 1.4 0.5
GATE and2  3 O=a*b;          PIN * NONINV 1 999 1.9 0.3 1.9 0.3
GATE or2   3 O=a+b;          PIN * NONINV 1 999 2.4 0.3 2.4 0.3
GATE xor2  5 O=a*!b+!a*b;    PIN * UNKNOWN 2 999 1.9 0.5 1.9 0.5
GATE xnor2 5 O=a*b+!a*!b;    PIN * UNKNOWN 2 999 2.1 0.5 2.1 0.5
GATE aoi21 3 O=!(a*b+c);     PIN * INV 1 999 1.6 0.4 1.6 0.4
GATE oai21 3 O=!((a+b)*c);   PIN * INV 1 999 1.6 0.4 1.6 0.4
GATE buf   1 O=a;            PIN * NONINV 1 999 1.0 0.0 1.0 0.0
GATE zero  0 O=CONST0;
GATE one   0 O=CONST1;
```

Figure 6.3 Modified technology mapping library.

3) Subsequently, we break the complex gates like *NOR*, *XNOR*, *AOI* and *OAI* gates into simple gates to gain access to intermediate signals. In the context of our case, the simple gates are 2-input *AND*, *OR*, *XOR* gates and *INV* gate. We refer to the resulting circuit as *rewritten circuit*.
4) Once the *rewritten circuit* is obtained, we use *BFS levelization* algorithm to levelize it. The achieved circuit is referred to as *levelized circuit*.
5) Lastly, we use the substitution method to verify whether the *levelized circuit* keeps the original functionality.

The reason that we break complex gates into simple gates (Step 3) is that the variable substitution method is very inefficient when complex gates are used. As Table 6.2 shows, the levelization step significantly increases the applicability of substitution verification method. Before levelization, it is difficult to verify even an 8-bit multiplier (It took over *2,000*s of CPU time and consuming over *15*GB memory on our server) using the variable substitution method. In contrast, it is much easier to verify larger multipliers after levelizing the synthesized and mapped circuits (See the CPU times shown in Table 6.2).

The last column of the table shows that the memory consumption to verify such circuits is still negligible.

It is imperative to include the levelization time into the total verification time because it is necessary to levelize the circuits prior to verification. That is, the total verification time is the sum of levelization time and variable substitution time. The remaining problem here is that the levelization times are much longer than running the substitution method itself. The efficiency of the proposed verification system will improve when the efficiency of levelization algorithm improves.

Unsigned multipliers	Number of gates	Levelization time (s)	Levelization mem (KB)	Verification time after levelization (s)	Verification mem after levelization (KB)
4-bit	128	0.01	7040	0.02	8240
8-bit	655	0.09	7292	0.05	8344
12-bit	1588	0.92	8136	0.11	8528
16-bit	2952	5.63	8684	0.20	8760
20-bit	4764	25.27	9376	0.31	9204
24-bit	6978	84.55	12116	0.46	9400
28-bit	9617	241.18	13172	0.62	10000
32-bit	16040	583.69	14440	0.84	10332
36-bit	16254	1272.36	15792	1.13	11212
40-bit	25240	2473.04	17376	2.45	17896

Table 6.2: CPU time comparison of circuits before and after levelization.

We should stress the need to map the synthesized circuit to a modified library (As mentioned in step 2). Such a mapping is needed since currently we cannot deal with circuits composed of arbitrary gates or higher level components. For example, the inclusion of *NAND3* gate into the library noticeably slows down our method. This applies to the inclusion of any other gate eliminated from the standard cell library. We have yet to figure out a concrete reason for this. It is one of our goal to allow the mapping of the synthesized circuits to arbitrary components.

6.2 Modified Levelization Algorithm based on Dijkstra's Algorithm

To improve the efficiency of levelization algorithm, we applied a modified Dijkstra's algorithm. The reason to use the Dijkstra's algorithm [53] is dictated by the similarities between levelization and shortesr-path problems. In the original Dijkstra's algorithm, beginning from the starting vertex, nodes with the smallest edge distance to a visited node cloud will be added to that cloud. The distances of previously added nodes will be updated using edge relaxation rule each time a new node is added to the cloud. In our case, we use the following assumptions.

- In a levelization iteration, a primary output bit will be selected as a starting point.
- Each edge (wire) in the graph has a constant weight of *1*.
- Each simple gate in the circuit represents a node in the graph and each node has an associated value that represents its distance to the current starting point.

In the following we refer to the modified algorithm as *Dijk_levelization*. Another reason of using the modified Dijkstra's algorithm is that it is much faster than the BFS implementation. The speed advantage of *Dijk_levelization* over the *BFS levelization* comes from the fact that each edge and node in the graph will be visited only once in each levelization step. The whole process keeps updating the distance dictionary rather than repeatedly adding and removing parent gates into and out of lists (which is a dominant part in *BFS levelization algorithm*). Based on these assumptions, the modified levelization algorithm is as follows.

```
Input: dictionary disGrpDict stores the nodes classified into groups
       according to their distances to the starting point, dictionary
       nodeDisDict stores distance of each node to the starting
       point, dictoinary adjacencyDict stores the output fanouts of
       each signal
Output: updated term dictionary disGrpDict
initialize candidate dictionary candiDict;
for each primary output bit $po_i$ do
    assign a level number (typically 0) to $po_i$;
    initialize each node's' distance to $po_i$ to negative infinity;
    while candiDict is not empty do
        put nodes that are closest to $po_i$ in candiDict to list targets;
        for each target in targets do
            get target's parents and do level relexation on them;
            update disGrpDict, nodeDisDict and candiDict;
            delete target from candiDict;
        end
    end
end
return disGrpDict;
```

Algorithm 6: *Dijk_levelization algorithm*.

The *Dijk_levelization* algorithm implements the BFS levelization of internal gates by postponing the substitution of a given variable as late as possible. It mimics the original *Dijkstra's* algorithm with the exceptions that all the edges have unit weights and the goal is finding the longest path from the starting point to each node. To clearly illustrate this process, we use the example circuit in Figure 6.3 which is comprised of six gates. We assume that the output of the *XOR* gate *g0* is the primary output bit. The levelization process starts from *g0*. Table 6.3 shows the updates of dictionaries used in the process. The dictionary *disGrpDict is* the final returned result.

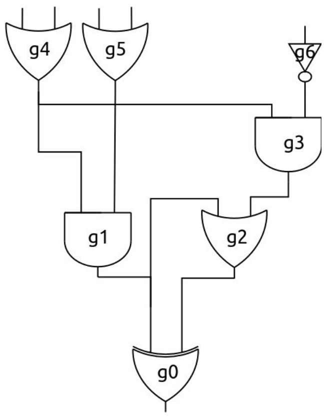


Figure 6.4: *Dijk_levelization example*.

visiting node	*candiDict {level : gates}*	*disGrpDict {level : gates}*	*nodeDisDict {gate : level}*
g3	*3 : g4, g5, g6*	*0 : g0, 1 : g2, 2 : g1, g3, 3 : g4, g5, g6*	*g0 : 0, g1 : 2, g2 : 1, g3 : 2, g4 : 3, g5 : 3, g6 : 3*
g1	*2 : g3, 3 : g4, g5*	*0 : g0, 1 : g2, 2 : g1, g3, 3 : g4, g5, -inf : g6*	*g0 : 0, g1 : 2, g2 : 1, g3 : 2, g4 : 3, g5 : 3, g6 : -inf*
g2	*2 : g1, g3*	*0 : g0, 1 : g2, 2 : g1, g3, -inf : g4, g5, g6*	*g0 : 0, g1 : 2, g2 : 1, g3 : 2, g4 : -inf, g5 : -inf, g6 : -inf*
g0	*1 : g2, 2 : g1*	*0 : g0, 1 : g2, 2 : g1, -inf : g3, g4, g5, g6*	*g0 : 0, g1 : 2, g2 : 1, g3 : -inf, g4 : -inf, g5 : -inf, g6 : -inf*
	0 : g0	*0 : g0 -inf : g1, g2, g3, g4, g5, g6*	*g0 : 0, g1 : -inf, g2 : -inf, g3 : -inf, g4 : -inf, g5 : -inf, g6 : -inf*

Table 6.3: Example of *Dijk_levelization procedure*.

We performed experiments on *Dijk_levelization* algorithm using the same hardware configuration and the same unsigned multiplier circuits (as used in testing

BFS_levelization algorithm). The results are shown in Table 6.4.

Unsigned multipliers	Number of gates	Levelization time (s)	Levelization mem (KB)	Verification time after levelization (s)	Verification mem after levelization (KB)
4-bit	128	0.03	7344	0.03	8228
8-bit	655	0.07	7848	0.07	8348
12-bit	1588	0.16	9196	0.12	8552
16-bit	2952	0.40	10344	0.20	8772
20-bit	4764	0.90	11532	0.32	9236
24-bit	6978	1.84	15328	0.47	9388
28-bit	9617	3.43	17404	0.63	10008
32-bit	16040	6.17	19956	0.84	10592
36-bit	16254	10.74	22864	1.11	11324
40-bit	25240	17.11	26484	2.34	18424

Table 6.4: CPU time comparison of circuits before and after *Dijk_levelization.*

As shown in Table 6.4, the CPU time performance of *Dijk_levelization* algorithm is more than *100*x times faster than that of *BFS_levelization* algorithm on average. The verification time of the levelized circuits are even better than those shown in Table 5.2.

In summary, by combining the proposed variable substitution method with the *Dijk_levelization* algorithm, we are able to verify large arithmetic circuits synthesized and mapped using ABC and the modified library. The CPU time performance and memory usage are both very good in our experiments.

6.3 More Experiments and Comparisons with Other Tools

6.3.1 Experiments on non-synthesized circuits

In this section, we present additional experiments which are performed on the same platform.

Signed Multipliers	Number of Gates	CPU Time (s) Unroll method (dict_sub)	Max Mem (KB) Unroll method (dict_sub)
2-bit	17	0.02	8216
3-bit	42	0.02	8220
4-bit	80	0.02	8220
5-bit	130	0.03	8232
6-bit	192	0.03	8252
8-bit	352	0.03	8296
10-bit	560	0.06	8384
12-bit	816	0.07	8460
16-bit	1472	0.12	8624
18-bit	1872	0.15	8772
20-bit	2320	0.17	8992
22-bit	2816	0.21	9180
24-bit	3360	0.25	9372
26-bit	3952	0.29	9560
28-bit	4592	0.34	9552
30-bit	5280	0.37	9872
32-bit	6016	0.44	10112
36-bit	7632	0.56	10480
40-bit	9441	0.69	11432
44-bit	11468	0.83	12164
48-bit	13633	1.00	13132
52-bit	16017	1.18	13756
56-bit	18593	1.36	14708
58-bit	19953	1.51	15136
60-bit	21361	1.6	15592
62-bit	22817	1.73	15988
64-bit	24319	1.82	16532
80-bit	37920	4.31	19228
96-bit	54720	6.43	26324
128-bit	97536	12.41	39648

Table 6.5: Verification time for signed multipliers.

The experiments on CAS adders are not provided in the table because the verification time is negligible (less than 1 second to verify a 128-bit adder). Table 7.1 shows the verification time for original singed multipliers generated by Genmult without any synthesis. It is an extension of experiments shown in Figure 4.9 and 4,10 in Section 4.4 from 64-bit to 128-bit operands. As an example, the CPU time for verifying a 128-bit multiplier is 12.41s with 39648KB maximum resident memory space used. The complexity of CPU time and memory usage are both linear.

Unsigned multipliers	Number of Gates	CPU Time (s) Unroll method dict_sub	Max Mem (KB) Unroll method dict_sub
4-bit	82	0.03	8224
8-bit	418	0.05	8340
12-bit	1010	0.10	8552
16-bit	1858	0.14	8936
20-bit	2962	0.23	9344
24-bit	4322	0.30	9704
28-bit	5938	0.41	10172
32-bit	7810	0.53	10512
36-bit	9938	0.68	11464
40-bit	12322	0.84	12224
44-bit	14962	1.02	13336
48-bit	17858	1.26	13968
52-bit	21012	1.45	15004
56-bit	24418	1.70	15972
60-bit	28082	1.97	17172
64-bit	32002	2.26	18292
80-bit	50242	3.68	25580
96-bit	72578	5.60	34644
128-bit	129538	10.99	39648

Table 6.6: Verification time for unsigned multipliers.

Table 7.2 shows the experimental results for unsigned multipliers, which are generated by BenGen. Comparing multipliers in Table 7.2 with the corresponding multipliers in Table 7.1, we find that the unsigned multipliers have larger number of gates but require less verification time. This is because the unsigned multipliers use the CSA structure while signed multipliers use the Wallace tree structure. The variable substitution method works faster on the CSA structure, possibly because that the signed multipliers need extra logic gates to implement 2's complement operations.

We compare our method with SMT tools, Z3 and CVC4; SAT tool of ABC; the symbolic algebra tool, Singular; and with Synopsys' Formality system.

Unsigned multipliers	Our method (s)	Z3 (s)	CVC4 (s)	ABC(SAT) (s)	Singular (s)	Formality (s)
4-bit	0.03	0.03	0.09	0.04	0.05	0.81
8-bit	0.05	16.55	42.63	11.66	TO	3.19
10-bit	0.08	1080.97	TO	127.37	TO	6.67
12-bit	0.10	TO	TO	UD	TO	108.1
14-bit	0.13	TO	TO	UD	TO	109.4
16-bit	0.14	TO	TO	UD	TO	111.2
64-bit	2.26	TO	TO	UD	TO	675.4
128-bit	10.99	TO	TO	UD	TO	TO

Table 6.7: Unsigned multipliers verification CPU time comparison with SMT, SAT/ABC, Singular and Formality.
(TO = timeout after 3600 sec, UD = UNDECIDED)

Figure 7.3 shows that our technique surpasses those tools in CPU time by several orders of magnitude. Other circuits could not be handled by these tools beyond just a small number of bits. Memory usage of these tools for the successful cases was comparable with ours.

6.3.2 Experiments on synthesized circuits

The following experiments are implemented to verify synthesized circuits. First we use ABC to synthesize the multipliers, and map the synthesized multipliers to our library (shown in Figure 6.3). The commands used to synthesize and map the circuits are:

abc 01> read research.genlib
abc 02> read multiplier.v
abc 03> resyn
abc 04> map
abc 05> write multiplier-syn-map.v

The command "*resyn*" is a combination of several synthesis commands include rewriting and structural hashing. It performs technology-independent rewriting and balances the circuit network. The synthesized and mapped multipliers are then levelized using *Dijk_levelization* algorithm described in Chapter 6. Finally, such pre-processed circuits are verified using the variable substitution method proposed in the thesis.

Firstly, we performed experiments on unsigned multipliers generated by BenGen.

Unsigned multipliers	Number of gates	Levelization time (s)	Levelization mem (KB)	Substitution time after levelization (s)	Substitution mem after levelization (KB)
4-bit	128	0.03	7344	0.03	8228
8-bit	655	0.07	7848	0.07	8348
12-bit	1588	0.16	9196	0.12	8552
16-bit	2952	0.40	10344	0.20	8772
20-bit	4764	0.67	14064	0.32	9236
24-bit	6978	1.12	20140	0.47	9388
28-bit	9617	1.76	22904	0.63	10008
32-bit	16040	2.62	25616	0.84	10592
36-bit	16254	3.74	29604	1.11	11324
40-bit	25240	5.15	33676	2.34	18424
64-bit	53990	21.12	83836	16.91	46840
80-bit	85090	41.74	116648	43.74	95316
96-bit	123444	71.63	176360	75.07	92888
128-bit	221112	167.85	318652	107.78	157604

Table 6.8: Levelization and verification of synthesized unsigned multipliers.

Table 7.4 shows the CPU times and memory usage of levelization and verification processes. The *verification time after levelization* is the CPU time used for verifying circuits that are synthesized, mapped and levelized. In the above table, levelization time and substitution time are comparable in the overall process.

Then we performed experiments on signed multipliers generated by Genmult. Initially, we performed the experiments exactly in the same steps as with unsigned multipliers. However, the experimental results were not good. It required more than one hour to verify a 16-bit signed multiplier. To solve this problem, we map the synthesized signed multipliers directly into a simple gate library that only contains 2-input *AND*, *OR*, *XOR* gates and *INV*, *BUF* gates (Previously, we mapped the synthesized circuits to library shown in Figure 6.3 then expended them to simple gates). The levelization time and variable substitution time used for the mapped circuits are shown in Table 7.5.

Signed multipliers	Number of gates	Levelization time (s)	Levelization mem (KB)	Substitution time after levelization (s)	Substitution mem after levelization (KB)
4-bit	76	0.03	7328	0.03	8304
8-bit	421	0.04	8044	0.11	8528
12-bit	1002	0.13	8716	0.22	8752
16-bit	1817	0.29	10688	0.18	8772
20-bit	2870	0.59	11788	0.29	8940
24-bit	4155	1.10	13288	0.43	9276
28-bit	5720	1.86	19486	0.61	9648
32-bit	7471	2.99	22108	0.82	9828
36-bit	9461	4.64	24720	1.05	10264
40-bit	11688	6.70	28260	1.30	11120
64-bit	30006	40.03	79136	3.47	15312
80-bit	46957	94.20	115600	5.66	21108
96-bit	67682	192.78	171816	8.29	27472
128-bit	120461	593.95	365488	13.91	40776

Table 6.9: Levelization and verification of synthesized signed multipliers.

The *verification time after substitution* in Table 7.5 is very good. It requires only 13.91s to verify a 128-bit singed multiplier. However, the levelization time increases in a quadratic manor. The conclusion drawn from the experiments are that by mapping a synthesized circuit to the simple gate library, we will obtain a circuit, which is harder to be levelized (compared with circuits achieved by being mapped to the complex gate library). However, once the levelized netlist is obtained, we can use the variable substitution method to verify the netlist easily.

We compare our method with winners of recent SMT competitions, including Boolector, Z3 and CVC4; SAT tool of ABC; the symbolic algebra tool, Singular; and with Synopsys' Formality system.

Unsigned multipliers	Our method (level+subst) (s)	Lingeling (s)	Minisat_blbd (s)	ABC (s)	Boolector (s)	Z3 (s)	CVC4 (s)	Formality (s)
4-bit	0.06	0	0	0.01	0	0.03	0.09	0.75
8-bit	0.14	4.4	62.75	11.66	7.18	16.55	42.63	2.9
12-bit	0.28	TO	1615.47	UD	2030.19	TO	TO	102.33
16-bit	0.6	TO	TO	UD	TO	TO	TO	TO
64-bit	175.54	TO	TO	UD	TO	TO	TO	TO
128-bit	4746.63	TO	TO	UD	TO	TO	TO	TO

Table 6.10: Unsigned multipliers verification CPU time comparison with SMT, SAT/ABC, Singular and Formality.
(TO = timeout after 3600 sec, UD = UNDECIDED)

Table 6.10 shows that our technique surpasses those tools in CPU time by several orders of magnitude. Our method consumes reasonable memory space when levelizing and verifying larger circuits.

CHAPTER 7

CONCLUSOINS AND FUTURE WORK

The methods proposed in this thesis work efficiently on circuits, which may contain fanout signals. The fanout signals that reconverge some levels later in the circuits may be useful as they create chance of algebraic cancellations. Considering the circuit signals as Boolean variables works together with algebraic cancellations to simplify the signature of the circuit. Circuits, which have very few fanout signals, are more diffcult to verify. When verifying such a circuit, the signature size, measured in the number polynomial terms, continues increasing until the last gate equations at the primary inputs are substituted. This is because each variable is used only once in the circuit; substituting such a variable cannot reduce the signature size at all. We plan to work around this problem by introducing redundant gates into circuits, creating so called "*vanishing polynomials*". These intentionally inserted gates should increase the term cancellation opportunity, while keeping the original functionality of the circuit intact.

Another issue is that currently we can not verify circuits synthesized with Design Complier (DC). It seems that Design Compiler has reduced the number of fanout signals, the number of reconvergent signals and minimizes redundancy. Even a circuit with hundreds of gates synthesized with DC makes our program crush due to the memory explosion. This may due to the reason stated here.

The future work that are undertaken by other group members in our lab including:

- Work on circuits synthesized with delay constraints. By applying delay constraints, the synthesized circuits may have structural redundancy that is useful for the proposed variable substitution method.
- Work on circuits mapped to complex gate library to better understand the effect of technology library on our method.
- Modify our method to extend it to sequential circuits by unrolling the circuit over a fixed number of time frames into a combinational circuit (bounded model).
- Work on how to use the proposed method to implement circuit debugging.

CHAPTER 8

CONTRIBUTIONS

This work described in this thesis has been done in collaboration with other members, Cunxi Yu and Walter Brown, of Professor Maciej Ciesielski's research team. During the research, I accomplished the following tasks:

- Wrote parsers for file format conversions.
- Studied symbolic computer algebraic theories and helped develop the theory.
- Developed the variable substitution algorithm. Some functions were borrowed with permission from a library written by Walter Brown.
- Tested the efficiency of the algorithm by running it on selected benchmark circuits.
- Studied how to efficiently use Singular to verify generated circuits, and compare the performance of Singular with our method.
- Used Formality (Synopsys) to verify the benchmark circuits, and compared the performance of Singular with our method.
- Used multi-process programming to improve the efficiency of variable substitution.
- Developed levelization method which helps apply our method to synthesized circuits.
- Developed Generate Parse Unroll (GPU) tool which integrates our algorithms into

one software.

- Contributed to the following papers:

  1) M. Ciesielski, W. Brown, C. Yu, D. Liu, *Verification of Gate-level Arithmetic Circuits by Function Extraction, DAC-2015*, submitted.

  2) C. Yu, M. Ciesielski, D. Liu, W. Brown, *Verification of Sequential Arithmetic Circuits, DAC-2015*, submitted.

  3) S. Ghandali, M. Ciesielski, C. Yu, D. Liu, *Fault Diagnosis and Logic Debugging of Arithmetic Circuits, DAC-2015*, submitted.

  4) M. Ciesielski, W. Brown, D. Liu, A. Rossi, *Function Extraction using Network Flow Model, interactive presentation/poster, Design Automation Conference, DAC-2014,* June 2014.

  5) M. Ciesielski, W. Brown, D. Liu, A. Rossi, *Function Extraction from Arithmetic Bit-level Circuits, IEEE Computer Society Annual Symposium on VLSI (ISVLSI),* 356 - 361, July 2014.

APPENDIX

KEY FUNCTION INPLEMENTATIONS AND SOFTWARE INTERFACE

Function determines term coefficient and monomial:

```
def determCoef(Term, outerCoef):
# determine coef:
numbers = '-123456789'
comb = Term.split('*',1)
if len(comb) == 1: # if is a single variable term or int
  try:
    coef = int(Term)*outerCoef
    Term = "1"
  except:
    if '-' in Term:
      coef = -1*outerCoef
      Term = comb[0].replace('-','')
    else:
      coef = outerCoef
else:
    tempCoef = comb[0]
    #determine coefficients:
    if not tempCoef[0] in numbers: #case a
      coef = outerCoef
    elif tempCoef[0] == '-' and not tempCoef[1] in numbers: #case -a
      coef = -1*outerCoef
      Term = Term.replace('-','')
    else:
      coef = int(tempCoef)*outerCoef
      Term = comb[1]
return (Term, coef)
```

Function implements fast substitution:

```
def and_term_substitute(and_term, var2sub, eqn_right):
'''substitute the var2sub variable in  and_term, and_term shall have no coefficient and
sign '-' '''
  dict_substitute = {}
  var_list1 = and_term.split('*')
```

```
 var_list1.remove(var2sub)
 if var_list1 == []:
   newExpr=eqn_right
 else:
   mult1 = '*'.join(var_list1)
   var_list2 = splitDNF2terms(eqn_right)
   for term in var_list2:
     tempTup = determCoef(term, 1)
     mult2 = tempTup[0]
     tempCoef = tempTup[1]
     tempTerm = BoolMult(mult2, mult1)
     if tempTerm in dict_substitute:
       coef = tempCoef+dict_substitute[tempTerm]
       if coef == 0 : del dict_substitute[tempTerm]
       else: dict_substitute[tempTerm] = coef
     else: dict_substitute[tempTerm] = tempCoef
   newExpr = ''
   for term in dict_substitute:
     if dict_substitute[term] < 0:
       if dict_substitute[term] == -1:
         newExpr += '-'+term
       else:
         newExpr += str(dict_substitute[term])+'*'+term
     elif dict_substitute[term] ==1:
       newExpr += '+'+term
     else:
       newExpr += '+'+str(dict_substitute[term])+'*'+term
 newExpr=newExpr.strip('+')
 return newExpr
```

Function recognizes primary output bits:

```
def initialDict(sigout):
 initialDict = {}
 initialDict['1'] = {'1':0}
 poList = splitDNF2terms(sigout)
 for po in poList:
   poComb = determCoef(po,1)
   coef = poComb[1]
   mono = poComb[0]
   subVars = mono.split('*')
   for var in subVars:
     if not var in initialDict: initialDict[var] = {}
     if mono in initialDict[var]:
       initialDict[var][mono] = initialDict[var][mono]+coef
     else:
       initialDict[var][mono] = coef
 return initialDict
```

The Generate Parse Unroll (GPU) software

Chapter 3 and Chapter 4 introduced the variable substitution verification method. Chapter 5 introduced the levelization algorithm that extends the application of the proposed verification method to synthesized circuits. In this chapter, we present the softeware GENERATE PARSE UNROLL, *GPU*, that integrates the two basic methods plus some other useful parsers and functions together. Figure 7.1 shows the main Graphic User Interface (GUI) of the software. It has five functions that are integrated.

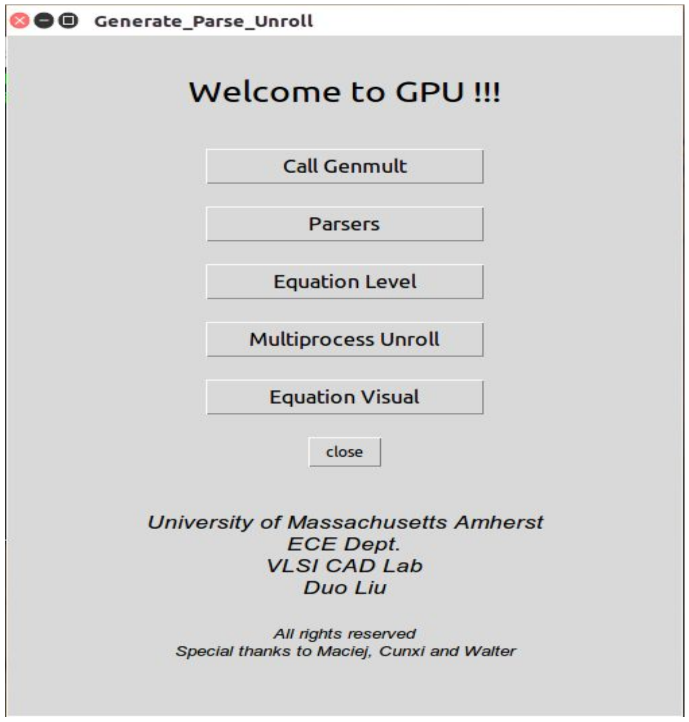
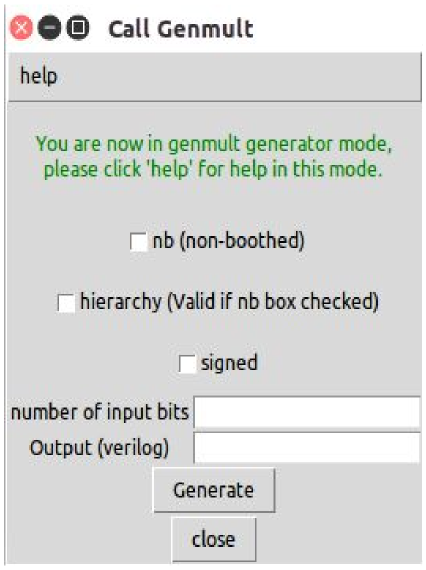


Figure A.1: GPU main user interface.

- **Call Genmult Module**

The *Call* Genmult module calls the Genmult program to generate multipliers and adders. Some bugs in the files generated by Genmult are also corrected. Currently it is the only module that generates benchmark circuits in the software. Other benchmark circuits used in this thesis are generated by BenGen.. The graphical view of this module is shown in Figure 7.2.


Call Genmult

help

You are now in genmult generator mode, please click 'help' for help in this mode.

- [ ] nb (non-boothed)
- [ ] hierarchy (Valid if nb box checked)
- [ ] signed

number of input bits

Output (verilog)

Figure A.2: Call_Genmult interface.

By setting the configurations, such as circuit structure and the number of input bits, user can use this module to generate benchmark multipliers.

- **Parsers Module**

The *Parsers* module converts file formats from one to another. Currently, the supported file formats include: equation files, structural verilog files, technology-mapped verilog files and files with format required by Singular. The graphical view of this module is shown in Figure 7.3.

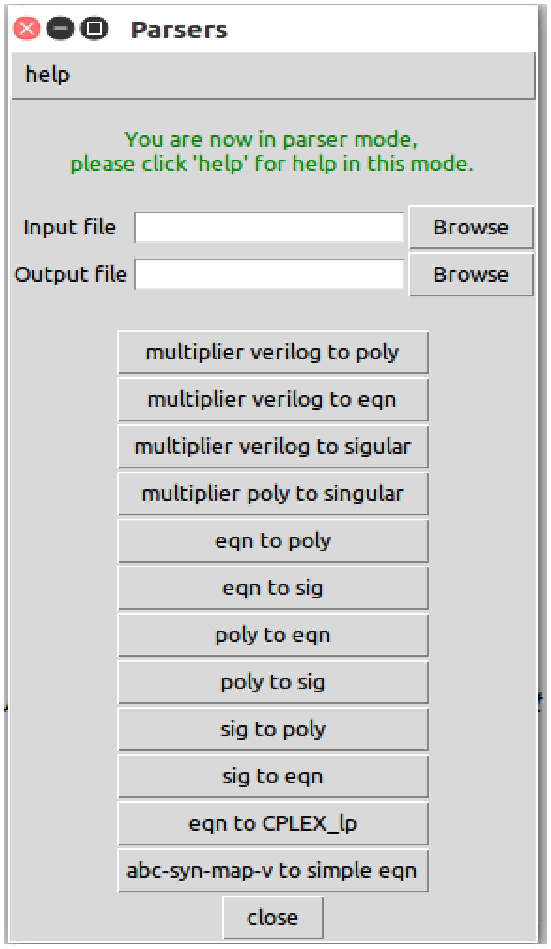

Parsers
help
You are now in parser mode,
please click 'help' for help in this mode.
Input file
Browse
Output file
Browse
multiplier verilog to poly
multiplier verilog to eqn
multiplier verilog to sigular
multiplier poly to singular
eqn to poly
eqn to sig
poly to eqn
poly to sig
sig to poly
sig to eqn
eqn to CPLEX_lp
abc-syn-map-v to simple eqn
close

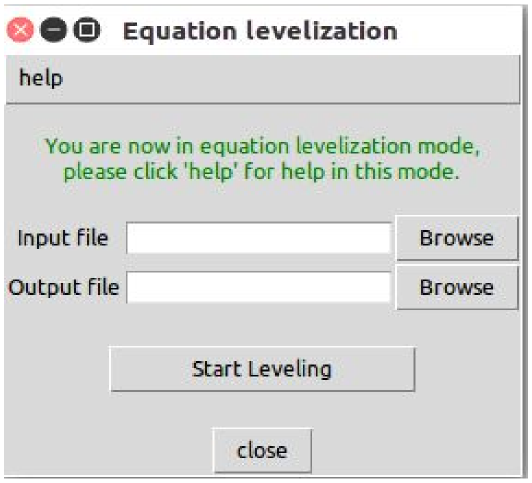
Figure A.3: Parsers interface.

In this module, user needs to specify one input file and one output file. By clicking the intended conversion, the software generates the required file. The Verilog file can be written in variant styles, currently the conversion from Verilog to other formats only supports the Verilog files generated by Genmult or BenGen.

The last conversion, "*abc-syn-map-v to simple eqn*" is used to expand complex gates synthesized by ABC. It maps the Verilog files into equation files containing only simple gates.

- **Levelization Module**

The levelization module accepts an equation file as input. This equation file must have output signature on the top, with gate equations following it. By clicking *Start Levelizing*, the module tries to recognize primary output bits in the first line, and assigns gate equations to proper level. The levelization time and memory usage will be reported on the terminal when levelization is done.



You are now in equation levelization mode, please click 'help' for help in this mode.

Figure A.4: Levelization interface.

- **Multiprocess Unroll Module**

This module takes the output of Levelization module as input, and computes the input signature with respect to the given equation file using the proposed variable substitution method. This module is called *Multiprocess Unroll*, because it can do variable substitution for two circuits simultaneously, one on each processor core. The graphical view of this module is shown in Figure 7.5.

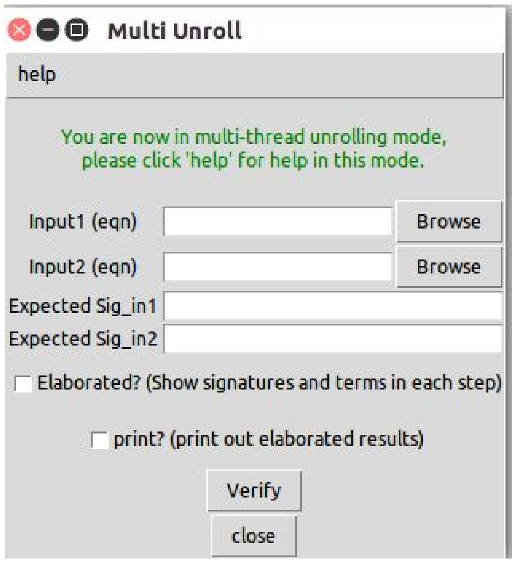

Multi Unroll
help
You are now in multi-thread unrolling mode, please click 'help' for help in this mode.
Input1 (eqn)
Browse
Input2 (eqn)
Browse
Expected Sig_in1
Expected Sig_in2
Elaborated? (Show signatures and terms in each step)
print? (print out elaborated results)
Verify
close

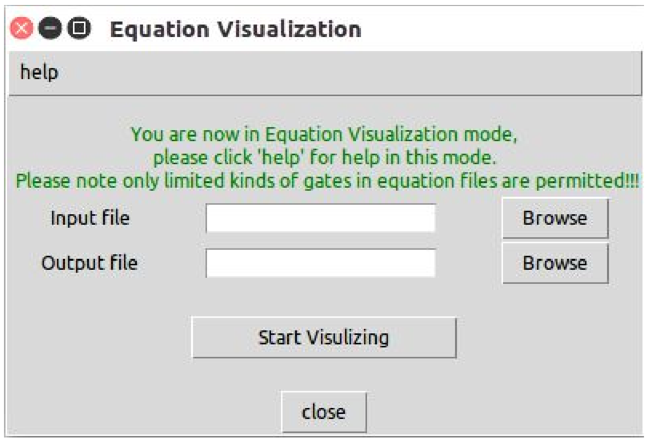
Figure A.5: Multiprocess Unroll interface.

There are three kinds of usage of this module. If one just wants to check whether the circuit implements the expected function, one needs to provide both the equation file and the expected input signature correspondingly. Specifically, *Input1* is related to *Expected Sig_in1*, and *Input2* is related to *Expected Sig_in2*. The module will automatically compute the difference between the computed input signature and the expected input signature. If the expected input signature is set to "*0*", the result will be the extracted function of the circuit.

The module can also be used to do the equivalence checking between two designs. In this case, the user should provide *Input1* and *Input2*, but leave the *Expected Sig_in1*and *Expected Sig_in2* blank. In this case, the module will compute the input signatures of both designs simultaneously. After that, the computed signatures will be compared to check the equivalence between two designs. The two check boxes allow user to access the intermediate information of the whole substitution process.

- **Equation Visulization Module**

This module also takes the output of Levelization step as input and generates a schematic diagram for the circuit. Currently, this module can only recognize 2-input *AND*, *OR*, *XOR* gates and *INV*, *BUF* gates. The graphical view of this module is shown in Figure 7.6.



You are now in Equation Visualization mode,
please click 'help' for help in this mode.
Please note only limited kinds of gates in equation files are permitted!!!

Figure A.6: *Equation Visualization* interface.

A diagram generated for a 3-bit signed multiplier is shown in Figure 7.7.

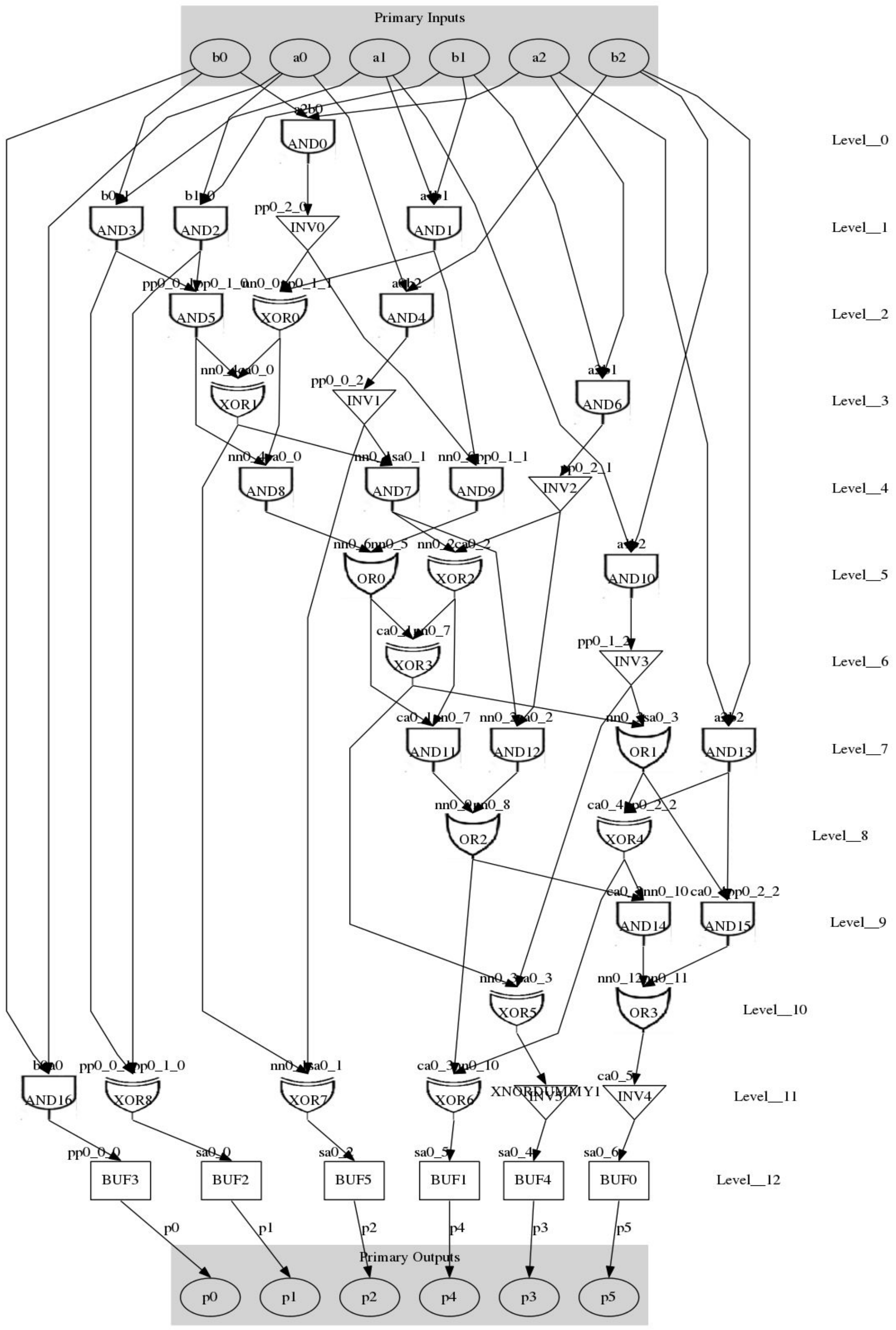


Figure A.7: *Equation Visualization* example.

BIBLIOGRAPHY

[1] Tahar, S., Cerny, E. and Song, X., "Formal Verification of Systems," *http://users.encs.concordia.ca/~tahar/coen6551/notes/1-intro-02.06.pdf*, May, 2002.

[2] Burkhardt, N., "Introduction to Functional Verification," *http://www.uni- frankfurt.de/39888300/21_7_2011_Burkhardt_Verification.pdf*.

[3] Ingalls, R. G., "Introduction to Simulation," *Simulation Conference (WSC), Proceedings of the 2011 Winter, IEEE*, pp. 1374-1388, Dec. 2005.

[4] Ciesielski, M., Brown, W., Liu, D. and Rossi, A., "Function Extraction from Arithmetic Bit-level Circuits," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 356-361, 2014.

[5] Dijkstra, E., "The Humble Programmer," *Classics in Software Engineering*, *Yourdon Press*, 1979.

[6] Bryant, R., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677-691, Aug. 1986.

[7] Bryant, R. and Chen, Y., "Verification of Arithmetic Circuits with Binary Moment Diagrams," *Design Automation, 1995. DAC '95. 32nd Conference on*, pp. 535-541, 1995.

[8] Ciesielski, M., Kalla, P. and Askar, S., "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *Computers, IEEE Transactions on*, vol. 55, no. 9, pp. 1188-1201, Sep. 2006.

[9] Segev, E., Goldshlager, S., Miller, H., Shua, O., Sher, O. and Greenberg, S., "Evaluating and comparing simulation verification vs. formal verification approach on block level design," *Electronics, Circuits and Systems, 2004. ICECS 2004. Proceedings of the 2004 11th IEEE International Conference on, IEEE*, pp. 515-518, Dec. 2004.

[10] Bjesse, P., "What is formal verification?," *ACM SIGDA Newslett.*, vol. 35, no. 24, Article 1, Dec. 2005.

[11] Sanghavi, A., "What is formal verification?," *http://www.eetasia.com/STATIC/PDF/201005/EEOL_2010MAY21_EDA_TA_01.pdf?SOURCES=DOWNLOAD*.

[12] Falkowski, B.J., "Equivalence Checking for Digital Circuits," *Potentials, IEEE*, pp. 21-23, Apr. 2004.

[13] Ciesielski, M. J., Kalla, P., Zheng, Zhihong and Rouzeyre, B., "Taylor Expansion Diagrams: A Compact, Canonical Representation with Applications to Symbolic Verification," *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pp. 285-289, 2002.

[14] Thornton, M. A., Drechsler, R. and Günther, W., "Logic Circuit Equivalence Checking Using Haar Spectral Coefficients and Partial BDDs," *VLSI Design*, vol. 14, no. 1, pp. 53-64, 2002.

[15] Emerson, E. A. and Clarke, E. M., "Characterizing Correctness Properties of Parallel Programs Using Fixpoints," *Proceedings of the 7th International Colloquium on Automata, Languages and Programming, Springer-Verlag*, pp. 169-181, 1980.

[16] Clarke, E. M., "Model Checking Overview," *http://www.cs.cmu.edu/~emc/15-398/ lectures/overview.pdf.*

[17] Bryant, R. E., "Symbolic Simulation—Techniques and Applications," *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 517-521, 1990.

[18] Jain, P. and Gopalakrishnan, G., "Efficient Symbolic Simulation-Based Verification Using the Parametric Form of Boolean Expressions," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pp. 1005-1015, Aug. 1994.

[19] Brzozowski, J. A. and Seger, C-J., "Correspondence Between Ternary Simulation and Binary Race Analysis in Gate Networks," *Proc. 13th Int. Colloq. Automata, Languages, Programming, L. Kott, Ed., Lecture Notes Comput. Sci.*, pp. 69-78, 1986.

[20] Seger, C.-J. H. and Bryant, R. E., "Model Checking Overview," *Simulation Conference (WSC), Proceedings of the 2011 Winter, IEEE*, pp. 1374-1388, Dec. 2011.

[21] Brand, D., "Verification of Large Synthesized Designs," *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pp. 534-537, Nov. 1993.

[22] Weaver, S., "Equivalence Checking," *http://gauss.ececs.uc.edu/Courses/c626/ lectures/SAT/Equivalence_ Checking_11_30_08.pdf.*

[23] Kuehlmann, A., "Dynamic Transition Relation Simplification for Bounded Property Checking," *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pp. 50-57, Nov. 2004.

[24] Hu, A. J., "Formal Hardware Verification with BDDs: An Introduction," *Communications, Computers and Signal Processing, 1997. 10 Years PACRIM 1987-1997 - Networking the Pacific Rim. 1997 IEEE Pacific Rim Conference on*, vol. 2, pp. 677-682, Aug. 1997.

[25] Lee, C. Y., "Representation of Switching Circuits by Binary-Decision Programs," *The Bell System Technical Journal,* vol. 38, no. 4, pp. 985-999, Jul. 1959.

[26] Akers, S.B., "Binary Decision Diagrams," *Computers, IEEE Transactions on*, vol. C-27, no. , pp. 509-516, 1978.

[27] Burch, J. R., "Using BDDs to Verify Multipliers," *Design Automation Conference, 1991. 28th ACM/IEEE*, pp. 408-412, Jun. 1991.

[28] Ashar, P., Ghosh, A and Devadas, S, "Boolean Satisfiability and Equivalence Checking Using General Binary Decision Diagrams," *Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings, 1991 IEEE International Conference on*, pp. 259-264, Oct. 1991.

[29] Brace, K., Rudell,R. and Bryant, R. E. , "Efficient Implementation of a BDD Package," *In Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 40-45, Jun. 1990.

[30] Ochi, H., Yasuoka, K. and Yajima, S. , "Breadth-first Manipulation of Very Large Binary-Decision Diagrams," *In Proceedings of the International Conference on Computer-Aided Design*, pp. 48-55, Nov. 1993.

[31] Somenzi , F, "CUDD: CU Decision Diagram Package ," http://gts.sourceforge.net, 2012.

[32] Yang, B., Chen, Y.-A., Bryant, R. E. and O'Hallaron, D. R., "Space- and Time-Efficient BDD Construction via Working Set Control," *Design Automation Conference 1998. Proceedings of the ASP-DAC '98. Asia and South Pacific*, pp. 423-432, Feb. 1998.

[33] Drechsler, R., Becker, B. and Ruppertz, S., "The K*BMD: A Verification Data Structure," *Design Test of Computers, IEEE*, vol. 14, no. 2, pp. 51-59, Apr. 2014.

[34] Lai, Y.-T. and Sastry, S., "Edge-valued Binary Decision for Multi-level Hierarchical Verification," *Design Automation Conference, 1992. Proceedings., 29th ACM/IEEE*, pp. 608-613, Jun. 1992.

[35] Cox, D., Little, J. and O'Shea D., "Ideals, Varieties and Algorithms," *New York : Springer-Verlag, ©1992.*, 1997.

[36] Lv, J., Kalla, P. and Enescu, F., "Efficient Grobner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 9, pp. 1409-1420, Spt. 2013.

[37] Kalla, P., "Leveraging Groebner Bases and SAT for Hardware/Software Verification," *http://www.birs.ca/workshops/2014/14w5101/files/kalla_birs_ talk.pdf*, Jan. 2014.

[38] Buchberger, B., "Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem Nulldimensionalen Polynomideal," *Ph.D.thesis, Univ. Innsbruck*, 1965.

[39] Faugere, J.-C., "A New Efficient Algorithm for Computing Groebner Bases (F4)," *Journal of Pure and Applied Algebra*, vol. 139, no. 13, pp. 61-88, 1999.

[40] Shekhar, N., Kalla, P. and Enescu, F., "Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 1320-1330, Jul. 2007.

[41] Wienand, O., Wedler, M., Stoffel, D., Kunz, W. and Greuel, G.-M., "An Algebraic Approach for Proving Data Correctness in Arithmetic Data Paths," *Computer Aided Verification*, pp. 473-486, Jul. 2008.

[42] Pavlenko, E., Wedler, M., Stoffel, D., Kunz, W., Dreyer, A., Seelisch, F. and Greuel, G., "Stable: A New QF-BV SMT Solver for hard Verification Problems Combining Boolean Reasoning with Computer Algebra," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1-6, Mar. 2011.

[43] Decker, W., Greuel, G.-M., Pfister, G. and Schonemann, H. , "*SIN-GULAR 3-1-6 — A computer algebra system for polynomial computations" Tech. Rep., [Software]*, *http://www.singular.uni-kl.de*, 2012.

[44] Basith, M.A., Ahmad, T., Rossi, A. and Ciesielski, M., "Algebraic Approach to Arithmetic Design Verification," *Formal Methods in CAD*, pp. 67-71, Oct. 2011.

[45] Ciesielski, M., Brown, W. and Rossi, A., "Arithmetic Bit-level Verification Using Network Flow Model," *Springer*, vol. LNCS 8244, pp. 327-343, 2013.

[46] Pruss, T., Kalla, P. and Enescu, F., "Equivalence Verification of Large Galois Field Arithmetic Circuits Using Word-Level Abstraction via GrÖBner Bases," *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pp. 152:1-152:6, 2014.

[47] Ciesielski, M., Kalla, P. and Askar, S., "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *IEEE Trans. on Computers*, vol. 55, no. 9, pp. 1188-1201, Sep. 2006.

[48] Ciesielski, M., Gomez-Prado, D., Ren, Q., Guillot, J. and Boutillon, E., "Optimization of Data-Flow Computation Using Canonical TED Representation," *IEEE Trans. on Computers*, vol. 28, no. 9, pp. 1321-1333, Sep. 2009.

[49] Genmult, *http://theory.cs.uvic.ca/dis/distribute.pl.cgi?package=GenMult.p*

[50] Time Complexity, *https://wiki.python.org/moin/TimeComplexity*

[51] Mishchenko, A et al., "ABC: A system for sequential synthesis and verification," *Retrievd from http://www. eecs. berkeley. edu/˜alanmi/abc,* 2007.

[52] Yu, Cunxi, "BenGen: Arithmetic Benchmark Generator," *http://people.umass.edu/ ycunxi/examples.html*, 2014.

[53] Dijkstra, E. W., "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematlk 1*, pp. 269-271, 1959.