Design Modification for Polynomial Formal Verification

Rolf Drechsler
University of Bremen/DFKI
Bremen, Germany
drechsler@uni-bremen.de

Alireza Mahzoon
University of Bremen
Bremen, Germany
mahzoon@informatik.uni-bremen.de

***Abstract*—With the rapid increase in the size and the complexity of digital circuits, the error rate in the design process is getting higher. In order to avoid the enormous financial loss due to the production of buggy circuits, using scalable formal verification methods is essential. The scalability of a verification method for a specific design is proven by showing that the method has polynomial space and time complexities. Unfortunately, not all verification methods have a polynomial complexity, particularly when it comes to the verification of complex designs.**

In this paper, we propose a novel design modification method for polynomial formal verification of complex designs. Complex designs consist of several smaller units that can be verified polynomially using a specific verification method. However, there is not a verification technique that verifies the whole design (including all units) in polynomial space and time. Our novel method first detects the suitable verification method for each design unit and then makes them visible to the verification process with minor design modifications. Thus, the verification can be carried out polynomially which was not possible with a single verification technique and without design modification before. The experimental results confirm the efficiency of our method for the verification of complex integer multipliers.

***Keywords*-polynomial verification; design modification; complexity; multiplier;**

I. Introduction

The size and complexity of digital circuits are increasing rapidly. Back in 1970, an Intel 4004 processor only had $2,250$ transistors. It could only support a limited number of instructions, and it was working at a very low frequency. However, the digital circuits nowadays are much larger, sometimes even consists of billions of transistors. Moreover, they are usually designed based on sophisticated algorithms, leading to fast but complex architectures. The big size and the high complexity of modern digital circuits make them extremely error-prone during the different design phases. For example, a mistake by the designer during the design of the *Register Transfer Level* (RTL) description or a failure in the synthesis process can lead to a buggy architecture. The implementation and production of these buggy designs cause a catastrophe, resulting in huge financial losses. This includes, e.g. the Intel Pentium bug in 1994, the New York blackout in 2003, and a design flaw in Intel's Sandy Bridge chipset in 2011.

Therefore, an important phase after the design of a digital circuit is to ensure its correctness. Formal verification and validation are two techniques to check the correctness of a digital circuit against its specification. In validation, it is accomplished through simulation; however, exhaustive simulation for the big designs is generally infeasible. On the other hand, formal verification takes advantage of rigorous mathematical reasoning to prove that a design meets all or parts of its specification [1], [2]. Several formal verification methods have been proposed to verify digital circuits. Every year, researchers come up with new verification methods to prove the correctness of specific types of circuits. These methods are usually fast and efficient in practice. However, a question remains unanswered for most of these approaches: Are they always scalable?

In order to correctly answer the question, we should first calculate the space and time complexity of verification methods. A verification approach is scalable if it has polynomial complexity, i.e. its space and time complexity is bounded by $O(n^m)$ where n is the number of circuit's inputs and m is a positive number. Recently, several works have been done in the field of polynomial formal verification of adders [3], [4], [5], multipliers [6], [7], *Arithmetic Logic Units* (ALUs) [8], totally symmetric functions [9], and tree-like circuits [10]. Unfortunately, polynomial formal verification is not always possible for complex designs due to two possible reasons: 1) the space or time complexity is exponential; thus, the method is not scalable, 2) the method is a heuristic; thus, it might report good results in practice but its complexity cannot be calculated. A verification method with exponential or unknown complexities is unreliable because it is not scalable, or its scalability (i.e. the growth of verification runtime and the memory usage with respect to the circuit's size) is unpredictable.

In this paper, we show that minor modifications in designs can make the polynomial formal verification possible. Digital circuits usually consist of several units, connected to perform a specific function. The polynomial verification of these circuits using only one verification method is not always possible. However, each unit inside the circuit can be individually verified with a formal method whose space and time complexity is polynomially bounded. The polynomial formal verification of each unit results in the polynomial formal verification of the whole design which was not

possible with only one method. However, the boundaries of design units are not always visible after the design process, as the design under verification is usually a flattened gate-level netlist without any hierarchical information. We show that some minor modifications in the design can make the boundaries visible. Thus, we can apply a polynomial formal verification method to each unit. As a case study, we focus on the integer multipliers as the polynomial formal verification of them is usually impossible. We depict that the design modification can make different stages of a multiplier visible to the verification process. Then, we use two formal verification approaches, i.e. *Symbolic Computer Algebra* (SCA) and *Binary Decision Diagram* (BDD) to verify the stages.

The remainder of the paper is structured as follows: The next section introduces the preliminaries needed in the paper. Section III discusses the challenges of polynomial formal verification for complex designs. In Section IV, we propose our novel design modification method for the polynomial verification. Section V presents the application of the proposed method for the formal verification of integer multipliers. The experimental results are reported in Section VI. Finally, Section VII concludes the paper.

II. Preliminaries

In this section, first, we introduce the general structure of a multiplier, which is our case study for the design modification. Then, we review the SCA- and BDD-based verification, which we later use for the verification of modified multipliers.

A. Multiplier Structure

Figure 1 shows the general structure of an integer multiplier consisting of three stages: *Partial Product Generator* (PPG), *Partial Product Accumulator* (PPA), and *Final Stage Adder* (FSA). The PPG stage generates partial products from the multiplier and the multiplicand inputs. Then, the PPA stage reduces the partial products by multi-operand adders and computes their sum. Eventually, the sum is converted to the corresponding binary output at the FSA [11], [12].

Several algorithms have been proposed to implement each stage of an integer multiplier. The architectures generated by them have some pros and cons in terms of design parameters, e.g. area, delay, power, and the number of wiring tracks. The designer can choose between different algorithms to achieve the design goal, e.g. minimizing the chip area. For example, Booth PPG [13] generates fewer partial products compared to Simple PPG; thus, it reduces the overall area of the multipliers with long operands. However, it has a higher design and logic complexity. As another example, the Wallace tree [14] and balanced delay tree [15] are two well-known algorithms for implementing the PPA stage. Wallace tree guarantees the lowest overall delay, but it has the largest number of wiring tracks. On the other hand, the balanced delay tree requires the smallest number of wiring tracks but suffers from the highest overall delay compared to other algorithms. As the last example, ripple carry adder and carry look-ahead adder are two algorithms for the implementation of the FSA. The ripple carry adder has the lowest area, but it suffers from a large delay. In contrast, the carry look-ahead adder has a much smaller delay, but it occupies more area. In the rest of the paper, we use the notation $[\boldsymbol{\alpha} \circ \boldsymbol{\beta} \circ \boldsymbol{\gamma}]$ to refer to a multiplier consisting of the stages: PPG α, PPA β, and FSA γ.

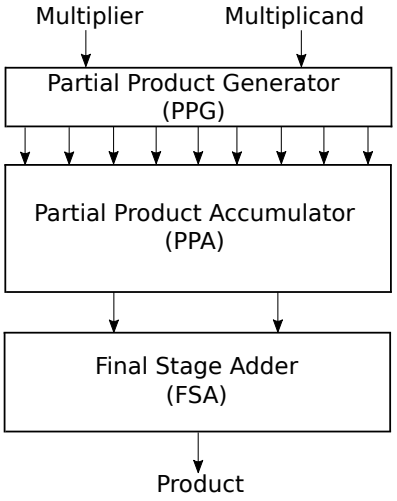


Figure 1: General multiplier structure

B. SCA-based Verification

Definition 1. *A* monomial *is the power product of variables in the following form:*

$$t = x_1^{\alpha_1} x_2^{\alpha_2} \dots x_n^{\alpha_n} \quad with \quad \alpha_i \in \mathbb{N}_0 \tag{1}$$

A monomial with a coefficient is called a Term*.*

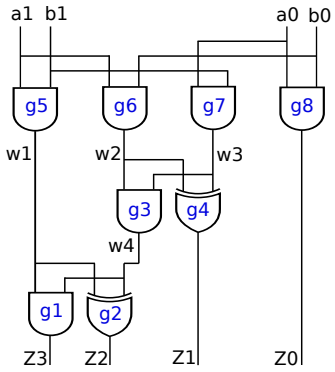
Definition 2. *A* polynomial *is a finite sum of monomials with coefficients in field k:*

$$f = \sum_j c_j t_j \quad with \quad c_j \in k \tag{2}$$

In SCA, the division is denoted by $p \xrightarrow{F} r$, where F is a set of polynomials and r is the remainder. For example, if $p = xy$, $f_1 = x - z$, and $f_2 = yz$, then $xy \xrightarrow{f_1} yz \xrightarrow{f_2} 0$. To perform the division of xy by f_1, first f_1 is multiplied by y to produce the same leading monomial xy as p, so $f_1 y = xy - yz$. Subsequently, the subtraction is performed, i.e. $p - (f_1 y) = xy - (xy - yz) = yz$, which is the result of the first division. Finally, yz is divided by f_2 to get remainder 0.

In SCA-based verification of arithmetic circuits, the gate-level netlist and the specification polynomial are given as inputs, and the task is to formally prove that the specification

polynomial and the arithmetic circuit are equivalent. The *specification polynomial* is a polynomial determining the function of an arithmetic circuit based on its inputs and outputs. For example, the specification polynomial for the 2-bit multiplier of Figure 2(a) is $SP = 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (2a_1 + a_0)(2b_1 + b_0)$ where $8Z_3 + 4Z_2 + 2Z_1 + Z_0$ describes the 4-bit output, and $(2a_1+a_0)(2b_1+b_0)$ indicates the multiplication of the 2-bit inputs.



(a) 2-bit multiplier

$$SP := 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0b_0)$$
$$SP \xrightarrow{p_{g_1}} SP_1 := 8w_1w_4 + 4Z_2 + 2Z_1 + Z_0 - (4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0b_0)$$
$$SP_1 \xrightarrow{p_{g_2}} SP_2 := 4w_1 + 4w_4 + 2Z_1 + Z_0 - (4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0b_0)$$
$$SP_2 \xrightarrow{p_{g_3}} SP_3 := 4w_1 + 4w_2w_3 + 2Z_1 + Z_0 - (4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0b_0)$$
$$SP_3 \xrightarrow{p_{g_4}} SP_4 := 4w_1 + 2w_2 + 2w_3 + Z_0 - (4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0b_0)$$
$$SP_4 \xrightarrow{p_{g_5}} SP_5 := 2w_2 + 2w_3 + Z_0 - (2a_1b_0 + 2a_0b_1 + a_0b_0)$$
$$SP_5 \xrightarrow{p_{g_6}} SP_6 := 2w_3 + Z_0 - (2a_0b_1 + a_0b_0)$$
$$SP_6 \xrightarrow{p_{g_7}} SP_7 := Z_0 - (a_0b_0)$$
$$SP_7 \xrightarrow{p_{g_8}} r := 0$$

(b) Backward rewriting steps

Figure 2: 2-bit multiplier and backward rewriting steps

The gates of an arithmetic circuit can be modeled as polynomials determining the relation between output and inputs. The polynomials of basic Boolean gates are as follows:

$$\begin{aligned} z = \neg a \Rightarrow p_g &:= z - 1 + a, \qquad & z = a \vee b \Rightarrow p_g &:= z - a - b + ab, \\ z = a \wedge b \Rightarrow p_g &:= z - ab, & z = a \oplus b \Rightarrow p_g &:= z - a - b + 2ab \end{aligned} \tag{3}$$

The polynomials in (3) are in the form of $p_g = x - tail(p_g)$ where x is the gate's output, and $tail(p_g)$ is a function based on the gate's inputs.

The gate polynomials for the 2-bit multiplier of Figure 2(a) are:

$$\begin{aligned} p_{g_1} &:= Z_3 - w_1w_4 & p_{g_5} &:= w_1 - a_1b_1 \\ p_{g_2} &:= Z_2 - w_1 - w_4 + 2w_1w_4 & p_{g_6} &:= w_2 - a_1b_0 \\ p_{g_3} &:= w_4 - w_2w_3 & p_{g_7} &:= w_3 - a_0b_1 \\ p_{g_4} &:= Z_1 - w_2 - w_3 + 2w_2w_3 & p_{g_8} &:= Z_0 - a_0b_0 \end{aligned} \tag{4}$$

Assume that the signals of an arithmetic circuit are ordered based on the reverse-topological order (i.e. from outputs toward inputs). The specification polynomial SP and the gate-level netlist are equivalent, iff the remainder of dividing SP by gate polynomials becomes zero. This division is known as Gröbner basis reduction. For the theory of Gröbner basis and its application to verification of arithmetic circuits we refer to [16], [17].

The steps of dividing SP by $p_{g_1}, \ldots, p_{g_8}$ for the 2-bit multiplier of Figure 2(a) are shown in Figure 2(b). The final remainder of the division is equal to zero, hence the multiplier is bug-free. Please note that all variables in the polynomials are Boolean. Thus, x^n can be replaced by x. Furthermore, for integer arithmetic circuits, dividing SP_i by a gate polynomial $p_{g_i} = x_i - tail(p_{g_i})$ is equivalent to *substituting* x_i by $tail(p_{g_i})$ in SP_i. For example, to obtain the result of the first division step in Figure 2(b), Z_3 can be substituted by w_1w_4 in SP. The process of dividing the specification polynomial by gate polynomials (or equivalently substituting gate polynomials in the specification polynomial) is called *backward rewriting*.

C. BDD-based Verification

Definition 3. *A* Binary Decision Diagram *(BDD) is a directed, acyclic graph. Each node of the graph has two edges associated with the values of the variables 0 and 1. A BDD contains two terminal nodes (leaves) that are associated with the values of the function 0 or 1.*

Definition 4. *An* Ordered Binary Decision Diagram *(OBDD) is a BDD, where the variables occur in the same order in each path from the root to a leaf.*

Definition 5. *A* Reduced Ordered Binary Decision Diagram *(ROBDD) is an OBDD that contains a minimum number of nodes for a given variable order. The ROBDD of a Boolean function is always unique.*

The ITE operator (If-Then-Else) is used to calculate the results of the logical operations in BDDs:

$$ITE(f, g, h) = (f \wedge g) \vee (\bar{f} \wedge h) \tag{5}$$

The basic binary operations can be presented using the ITE operator:

$$\begin{aligned} f \wedge g &= ITE(f, g, 0), \\ f \vee g &= ITE(f, 1, g), \\ f \oplus g &= ITE(f, \overline{g}, g), \\ f \odot g &= ITE(f, g, \overline{g}), \\ \bar{f} &= ITE(f, 0, 1) \end{aligned} \tag{6}$$

ITE can be also used recursively in order to compute the results:

$$ITE(f, g, h) = ITE(x_i, ITE(f_{x_i}, g_{x_i}, h_{x_i}), ITE(f_{\overline{x}_i}, g_{\overline{x}_i}, h_{\overline{x}_i})) \tag{7}$$

where f_{x_i} ($f_{\overline{x}_i}$) is the positive (negative) cofactor of f with respect to x_i, i.e., the result of replacing x_i by the value 1 (0).

Algorithm 1 If-Then-Else (ITE)

Input: f, g, h BDDs
Output: ITE BDD

```
1: if terminal case then
2:     return result
3: else if computed-table has entry {f, g, h} then
4:     return result
5: else
6:     v = top variable for f, g, or h
7:     t = ITE(f_{v=1}, g_{v=1}, h_{v=1})
8:     e = ITE(f_{v=0}, g_{v=0}, h_{v=0})
9:     R = FindOrAddUniqueTable(v, t, e)
10:    InsertComputedTable({f, g, h}, R)
11:    return R
```

The algorithm for calculating ITE operations is presented in Algorithm 1. The result is computed recursively based on Eq. (7) in this algorithm. When calculating the results of ITE operations for the f, g, h BDDs, the arguments for subsequent calls to the ITE subroutine are the sub-diagrams of f, g and h.

In order to formally verify a circuit, we need to have the BDD representation of the outputs. Symbolic simulation helps us to obtain the BDD for each primary output. During a simulation, an input pattern is applied to a circuit, and the resulting output values are checked to see whether they match the expected values. On the other hand, symbolic simulation verifies a set of scalar tests (which usually cover the whole input space) with a single symbolic test. Symbolic simulation using BDDs is done by generating corresponding BDDs for the input signals. Then, starting from primary inputs, the BDD for the output of a gate (or a building block) is obtained using the ITE algorithm. This process continues until we reach the primary outputs. Finally, the output BDDs are evaluated to see whether they match the BDDs of the circuit.

III. Challenges of Formal Verification

Despite the rapid progress of formal verification methods, it is not possible to prove the correctness of many digital circuits in polynomial time and space. It is due to the fact that either 1) the formal method has an exponential time and space complexity in theory and practice or 2) it is a heuristic; thus, it is impossible to prove the polynomial behavior in theory in spite of the good experimental results.

Integer multipliers are good examples of circuits whose polynomial formal verification is usually not possible. Here, we summarize the results of applying some well-known verification methods to multipliers:

1) It has been proven that the size of BDDs for the outputs of a multiplier grows exponentially with respect to the size of the multiplier [18]. Therefore, BDD-based method cannot support the verification of integer multipliers with big input sizes in practice.
2) The run-time of verifying an integer multiplier using SAT-based verification method increases exponentially with respect to the size of the multiplier. Thus, similar to BDD-based method, SAT-based verification fails to prove the correctness of large multipliers.
3) It has been proven in [6] and [7] that the word-level polynomial verification of structurally simple multipliers whose second and third stage are only made of half-adders and full-adders is possible using *BMD and SCA-based techniques, respectively. Moreover, several SCA-based verification methods have tried to come up with some heuristics in order to attack the hard problem of verifying structurally complex multipliers [19], [20], [21], [22], [23], [24]. They report good result in practice; However, proving the polynomial behavior of them is impossible in theory.

The example of verifying an integer multiplier clearly shows that polynomial formal verification of complex circuits using only one verification technique is not always possible. However, complex circuits usually consist of smaller units that can be verified polynomially. For instance, a structurally complex multiplier can be verified in polynomial time and space if we use the suitable verification method for each stage (see Figure 1):

- *BMD and SCA-based methods can polynomially prove the correctness of the first and the second stages of a multiplier.
- BDD-based verification has polynomial time and space complexity for the final stage of a multiplier (i.e. final stage adder).

Thus, choosing the right verification method for each unit results in the successful polynomial formal verification of the whole design. However, in order to consider each unit independently during the verification, the design hierarchies should be available, which is not always the case. The circuit under verification is usually a flattened gate-level netlist without any information about the hierarchies and the boundaries of the units. This makes the separate verification of each unit impossible in practice.

In order to overcome this challenge, we come up with a new approach that includes minor design modifications. The modifications make the boundaries of the units inside the circuit visible to the verification process. As a result, we can verify each unit independently using a suitable verification method.

IV. Design Modification Method

In this section, we present our novel design modification technique for polynomial formal verification of complex circuits. Our technique consists of two main phases: Determining the suitable verification method for each unit and

modifying the design in order to make the units visible to the verification process.

A. Determining Suitable Verification Methods

The first phase of our design modification method is the determination of the suitable verification approach for each individual unit. To do so, we iterate through all design units and assign a verification method based on the type of the unit. The space and time complexity of the assigned verification method should be polynomial. Please note that if a group of connected units can be verified together, we assign a verification method to the whole group, i.e. the entire group can be verified at the same time using the determined method.

Figure 3 shows a complex circuit consisting of six units. The units are depicted with U_i, where $1 \leqslant i \leqslant 6$. To perform the first phase of the design modification, we iterate through the units and assign the suitable verification method G_i to them. As the units U_2 and U_3 are connected and can be verified together, we consider them as a group and assign a verification method G_2 to the group. Similarly, the units U_4 and U_5 form a group.

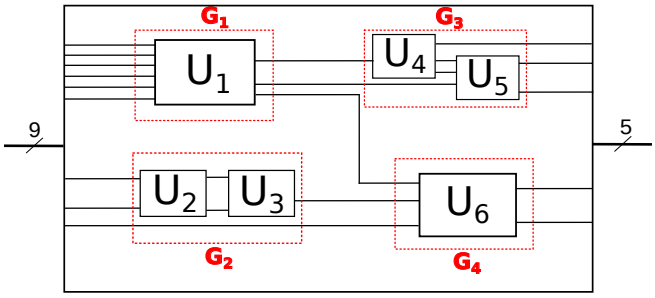


Figure 3: Assigning the suitable verification methods to the design units

B. Modifying Design for Verification

After determining the suitable verification method for each unit or group of units, we should make them visible to the verification process. This phase consists of two main steps:

- The outputs of each unit (or group of units) are set as the new outputs of the circuit. We call these outputs *Verification Outputs* (VO) as they are only used during the verification. If the outputs of a unit are the *Primary Outputs* (POs) of the circuit, we do not need new VOs since we can directly use POs for the verification.
- A multiplexer is added to the inputs of the units (or group of units). The first input of the multiplexer is connected to the inputs of the unit. The second input is connected to new inputs. We call these new inputs *Verification Inputs* (VI) as they are only used during the verification process. If the inputs of a unit are the *Primary Inputs* (PIs) of the circuit, we do not need new VIs since we can directly use PIs for the verification. We use a single *select* signal for all multiplexers, which is called *Verification Mode* (VM).

In the modified circuit, if the VM is set to zero (0) the circuit is working in the normal mode. Thus, PIs and POs are the only valid inputs and outputs of the circuit, and the values of VIs and VOs do not have any role in the function of the circuit. On the other hand, if the VM is set to one (1), the circuit enters the verification mode. In the verification mode, VIs and VOs are the valid inputs and outputs of the design. Therefore, if we apply values to VI_i, which is the verification input of the unit U_i, we get the output values in VO_i. Please note that during the verification, each unit or group of units is totally isolated from other units, and it can verified be separately.

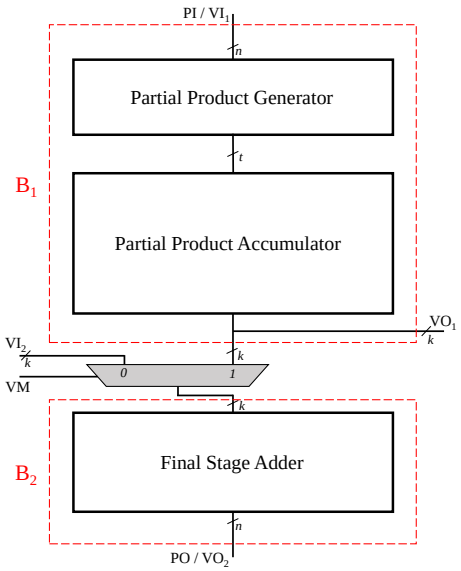


Figure 4: Modified multiplier for the formal verification

Figure 4 shows the application of the design modification approach on an integer multiplier. During the first phase (i.e. determining a suitable verification method), we realize that the first and the second stages of the multiplier are connected and can be verified together using the SCA-based verification method. Thus, we consider them as a group of units. The final stage of the multiplier is an adder, and it can be verified polynomially using the BDD-based method. Now, we need to make the group of the first and the second stages B_1 as well as the final stage B_2 visible to the verification process. In the second phase of the design modification, we set the outputs of B_1 as the new outputs of the circuit VO_1. Since the outputs of B_2 are connected to the POs, we do not need new outputs for VO_2. The inputs of B_1 are the PIs; thus, no multiplexer is required and we can directly use them as VI_1. We put a multiplexer in the inputs of B_2 and connect the new inputs VI_2 to its second

input.

V. Modified Multipliers Verification

In this section, we show how the modified multiplier is now verified polynomially using two verification methods.

First, the VM signal is set to one (1) in order to enter the verification mode. We start with the verification of B_1 whose inputs (VI_1) and outputs (VO_1) are now accessible in the verification mode: It has been shown that SCA-based verification proves the correctness of the first and the second stages of a multiplier polynomially for almost all architectures, including simple partial product generator and Booth partial product generator for the first stage, and array, Wallace tree, and Dadda tree for the second stage of the multiplier [7].

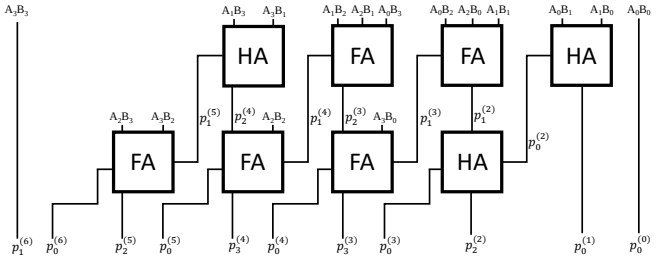


Figure 5: The first and the second stages of a multiplier

Figure 5 depicts the first and the second stages of a multiplier. The first stage has a simple partial product generator architecture. To simplify the design and avoid confusion, we have omitted the AND gates and shown their outputs with A_iB_i, where A_i and B_i are VI_1. The second stage of the multiplier has an array structure, where the multi-operand adders (half-adders and full-adders) are used to reduce the partial products. The partial products are named p_i^w, where w is the weight or significance. Before starting the SCA-based verification, we determine the specification polynomial. The outputs of the circuit (i.e. VO_1) are the partial products that their weights are available. A partial product p_i^w appears as the word-level form $2^w p_i^w$ in the specification polynomial. Thus, the SP for Figure 5 is as follows:

$$SP := 2^6 p_1^6 + 2^6 p_0^6 + 2^5 p_2^5 + 2^5 p_0^5 + 2^4 p_3^4 + \cdots + p_0^0 - (8A_3 + 4A_2 + 2A_1 + A_0) \times (8B_3 + 4B_2 + 2B_1 + B_0) \quad (8)$$

Starting from VO_1, the gate polynomials are substituted in SP step by step until we reach VI_1 (see Section II-B for more details). If we get zero remainder, B_1 (i.e. the first and the second stages of the multiplier) is correct; otherwise, it is buggy. since B_1 is mostly made of half-adders and full-adders, we can use the half-adder's and full-adder's polynomials during backward rewriting:

For example, $2^6 p_0^6 + 2^5 p_2^5$ is substituted directly with $2^5 A_2 B_3 + 2^5 A_3 B_2 + 2^5 p_1^5$; thus, we avoid the large intermediate polynomials resulted from the substitution of gates's polynomials.

Theorem 1. *The SCA-based verification of the first and the second stage of a multiplier has quadratic $O(n^2)$ space complexity and quartic $O(n^4)$ time complexity with respect to the number of input bits.*

Proof: A multiplier receives two inputs with n bits. The output of the second stage consists of two numbers with a maximum of $n-1$ bits, which should be added together by the final stage adder. Therefore, the specification polynomial (SP_i) has a maximum of $2n-2+n^2$ monomials (see Eq. (9)), where $2n-2$ is the maximum size for the word-level description of the output, and n^2 is the size of the polynomial obtained by multiplying the word-level description of the inputs.

$$SP := 2^{n-2} p_1^{(n-2)} + 2^{n-2} p_0^{(n-2)} + 2^{n-3} p_1^{(n-3)} + \cdots + p_0^{(0)} - (2^{n-1} A_{n-1} + \cdots + A_0) \times (2^{n-1} B_{n-1} + \cdots + B_0) \quad (9)$$

The total number of half-adders and full-adders in the second stage of a multiplier is $O(n^2)$. The substitution of each half-adder and full-adder polynomial increases the size of the current polynomial (SP_i) by zero and one monomial, respectively. Thus, after substituting all half-adders and full-adders, the size of the current monomial increases by a maximum of n^2 monomials. The first stage of the multiplier consists of AND gates whose polynomials are in the form A_iB_j. The substitution of each AND gate polynomial reduces the size of the current polynomial by two until zero remainder is obtained. As a result, the space complexity of verifying the first and second stages of a multiplier is $O(n^2)$.

In each step of backward rewriting, we first search the current polynomial for the proper variable for the substitution. The maximum size of the specification polynomial is $2n-2+n^2$ and it is increased by one during the backward rewriting of the second stage. Thus, the computational complexity of the backward rewriting for the second stage equals:

$$O(\sum_{i=0}^{n^2-1} (2n-2+n^2+i)) = O(n^4) \quad (10)$$

The size of the current polynomial is decreased by two during the backward rewriting of the first stage. Since there are n^2 AND gates in the first stage, the computational complexity is equal to:

$$O(\sum_{i=0}^{n^2-1} (2n-2+2n^2-2i)) = O(n^4) \quad (11)$$

As a result, the total time complexity of verifying the first and second stages of a multiplier is $O(n^4)$.

■

Table I: Results of verifying different multiplier architectures

Benchmark	Size	#Gates	Run-times (seconds)						
			Proposed method	Commercial	[25]	[17]	[26]	[27]	[19]	[23]
$SP \circ BD \circ KS$	16×16	2,101	0.01	50.00	TO	TO	TO	TO	TO	0.09
$BP \circ WT \circ CS$		1,821	0.02	47.00	TO	TO	TO	TO	TO	0.14
$SP \circ DT \circ LF$	32×32	8,046	0.11	TO	TO	TO	TO	TO	64.62	0.38
$SP \circ WT \circ CL$		12,066	0.14	TO	TO	TO	TO	TO	1,045.89	0.87
$SP \circ BD \circ KS$		8,577	0.11	TO	TO	TO	TO	TO	TO	0.76
$SP \circ AR \circ CK$		7,780	0.12	TO	TO	TO	TO	TO	TO	0.28
$BP \circ AR \circ RC$		6,314	0.26	TO	2.90	TO	0.02	TO	51.19	0.81
$BP \circ CT \circ BK$		5,766	0.27	TO	TO	TO	TO	TO	227.41	1.39
$BP \circ OS \circ CU$		7,357	0.25	TO	TO	TO	TO	TO	TO	1.30
$BP \circ WT \circ CS$		6,640	0.27	TO	TO	TO	TO	TO	TO	1.02
$SP \circ DT \circ LF$	64×64	32,680	1.72	TO	TO	TO	TO	TO	2,105.74	5.09
$SP \circ WT \circ CL$		52,083	1.84	TO	TO	TO	TO	TO	TO	16.53
$SP \circ BD \circ KS$		34,065	1.77	TO	TO	TO	TO	TO	TO	12.05
$SP \circ AR \circ CK$		31,944	1.71	TO	TO	TO	TO	TO	TO	3.07
$BP \circ AR \circ RC$		24,442	4.54	TO	37.18	TO	0.09	TO	882.52	14.36
$BP \circ CT \circ BK$		21,872	4.57	TO	TO	TO	TO	TO	1,729.33	28.53
$BP \circ OS \circ CU$		26,821	4.55	TO	TO	TO	TO	TO	TO	23.73
$BP \circ WT \circ CS$		24,830	4.59	TO	TO	TO	TO	TO	TO	41.48
$SP \circ WT \circ BK$	128×128	131,683	16.38	TO	TO	TO	TO	TO	TO	142.22
$SP \circ DT \circ LF$		131,297	17.04	TO	TO	TO	TO	TO	TO	153.60
$SP \circ WT \circ BK$	256×256	526,520	98.07	TO	TO	TO	TO	TO	TO	3,773.21
$SP \circ DT \circ LF$		525,531	98.58	TO	TO	TO	TO	TO	TO	5,622.45
$SP \circ WT \circ BK$	512×512	2,103,610	827.39	TO	TO	TO	TO	TO	TO	67,493.30
$SP \circ DT \circ LF$		2,101,205	836.12	TO	TO	TO	TO	TO	TO	114,257.87

Stage 1 ⇒ **SP**: Simple partial product generator **BP**: Booth partial product generator TO: Time-Out (150 hrs)
Stage 2 ⇒ **AR**: Array **BD**: Balanced delay tree **DT**: Dadda tree **WT**: Wallace tree **CT**: Compressor tree **OS**: Overturned-stairs tree
Stage 3 ⇒ **RC**: Ripple carry **BK**: Brent-Kung **LF**: Ladner-Fischer **CL**: Carry look-ahead **KS**: Kogge-Stone **CK**: Carry-skip **CS**: Carry select **CU**: Conditional sum

Now, we focus on the verification of B_2 whose inputs (VI_2) and outputs (VO_2) are accessible in the verification mode: BDD-based method reports very good results when it comes to the verification of adders. In order to prove the correctness of the final stage adder, we first apply the input BDDs to VI_2. Then, the BDDs for the outputs of gates are computed using ITE operation (see Section II-C for more details). This process continues until we reach VO_2 and obtain the BDDs for the outputs. Finally, the output BDDs are evaluated to see whether they match the BDDs of a correct adder.

Theorem 2. *The BDD-based verification of the final stage adder has polynomial space and time complexity with respect to the number of input bits.*

Recently, it has been proven in [3], [4], [5] that the polynomial formal verification of various adder architectures, including ripple carry adder, conditional sum adder, carry look-ahead adder, and parallel prefix adders is possible using BDD-based verification.

The polynomial formal verification of B_1 and B_2 in the modified multiplier of Figure 4 results in the polynomial formal verification of the whole circuit, which was not possible by using only one verification method and without design modification.

VI. Experimental Results

The efficiency of our design modification method is evaluated using a wide variety of structurally complex multiplier architectures. The multipliers with 16×16, 32×32, and 64×64 sizes have been generated with the AOKI generator [28]. This generator can build multiplier architectures only up to 64 bits per input. Therefore, we have generated multipliers up to 512 input bits using our own multiplier generator [29]. Then, the generated multipliers are modified based on Figure 4 in order to make the stages visible to the verification process. Moreover, the SCA-based and the BDD-based methods have been implemented in C++. The experiments have been carried out on an Intel(R) Xeon(R) CPU E3-1270 v3 3.50 GHz with 32 GByte of main memory.

In Table I, we report the results of verifying different modified multiplier architectures in the verification mode. The *Time-Out* (TO) has been set to 150 hours for all experiments. The first column of Table I presents the architecture of the multiplier based on its stages (see abbreviations below the table). The second column *Size* shows the size of the multiplier based on the input bits before the design modification. The number of gates for each architecture is given in the third column *#Gates*.

The fourth column of Table I reports the run-time of our proposed method. The remaining columns present the run-times of the most recent state-of-the-art formal verification methods. As can be seen, our approach can verify all multipliers with different architectures and sizes. It outperforms all the existing state-of-the-art formal verification methods by several orders of magnitude.

The run-times of seven state-of-the-art techniques are reported in the table: While *Commercial* reports the run-

times of the commercial verification tool Onespin, the remaining subcolumns give the run-times of some of the most recent SCA verification approaches. Please note that these approaches work on unmodified multipliers. The commercial tool only verifies 16×16 multipliers. The verification methods of [25], [17], [26], [27] only verify a few architectures and time-out for the rest. The proposed method in [19] supports the verification of more architectures. Finally, RevSCA 2.0 [23] verifies all multiplier architectures; however, its run-time is huge especially for the architectures bigger than 128×128 input sizes.

VII. Conclusion

In this paper, we have proposed a novel design modification method for polynomial formal verification of complex designs. We showed that some minor changes in the design can make the units visible to the design process. Then, each unit can be verified independently using a suitable formal verification method. Thus, we can prove the correctness of complex circuits in polynomial time and space, while it was not possible with only one verification method and without design modification. We particularly applied our method to the multiplier architectures and proved that polynomial formal verification of complex multipliers becomes possible. The experimental results showed that our method allows for verification of a wide variety of multiplier architecture with more than 2 million gates.

Acknowledgment

This work was supported by DFG within the Reinhart Koselleck Project *PolyVer* (DR 287/36-1).

References

[1] R. Drechsler, *Advanced Formal Verification*. Kluwer Academic Publishers, 2004.

[2] R. Drechsler, *Formal System Verification: State-of the-Art and Future Trends*. Springer, 2017.

[3] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *DDECS*, 2021, pp. 99–104.

[4] A. Mahzoon and R. Drechsler, "Late breaking results: Polynomial formal verification of fast adders," in *DAC*, 2021, pp. 1376–1377.

[5] A. Mahzoon and R. Drechsler, "Polynomial formal verification of prefix adders," in *ATS*, 2021, pp. 85–90.

[6] M. Keim, R. Drechsler, B. Becker, M. Martin, and P. Molitor, "Polynomial formal verification of multipliers," *Formal Methods in System Design: An International Journal*, vol. 22, no. 1, pp. 39–58, 2003.

[7] M. Barhoush, A. Mahzoon, and R. Drechsler, "Polynomial word-level verification of arithmetic circuits," in *MEMOCODE*, 2021, pp. 1–9.

[8] R. Drechsler, A. Mahzoon, and L. Weingarten, "Polynomial formal verification of arithmetic circuits," in *ICCIDE*, 2021, pp. 457–470.

[9] R. Drechsler and C. Dominik, "Edge verification: Ensuring correctness under resource constraints," in *SBCCI*, 2021, pp. 1–6.

[10] R. Drechsler, "Polynomial circuit verification using BDDs," in *ICEECCOT*, 2021, pp. 466–483.

[11] R. Zimmermann, "Binary adder architectures for cell-based VLSI and their synthesis," Ph.D. dissertation, Swiss Federal Institute of Technology, 1997.

[12] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. A. K. Peters, Ltd., 2001.

[13] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[14] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, 1964.

[15] D. Zuras and W. H. McAllister, "Balanced delay trees and combinatorial division in VLSI," *IEEE Journal of Solid-State Circuits*, vol. 21, no. 5, pp. 814–819, 1986.

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

[17] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017, pp. 23–30.

[18] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.

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

[20] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019, pp. 28–36.

[21] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *ICCAD*, 2018, pp. 129:1–129:8.

[22] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.

[23] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA-2.0: SCA-based formal verification of non-trivial multipliers using reverse engineering and local vanishing removal," *TCAD*, 2021, early access.

[24] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.

[25] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.

[26] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on and-inverter graphs," *TCAD*, vol. 37, no. 9, pp. 1907–1911, 2017.

[27] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *DATE*, 2018, pp. 1556–1561.

[28] "Arithmetic module generator based on ACG," available at https://www.ecsis.riec.tohoku.ac.jp/topics/amg/i-amg, 2019.

[29] A. Mahzoon, D. Große, and R. Drechsler, "GenMul: Generating architecturally complex multipliers to challenge formal verification tools," in *Recent Findings in Boolean Techniques*, R. Drechsler and D. Große, Eds. Springer, 2021, pp. 177–191.