Polynomial Formal Verification of Prefix Adders

Alireza Mahzoon[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{mahzoon,drechsle}@informatik.uni-bremen.de

***Abstract*—Nowadays, prefix adders are widely used in different designs and applications due to their flexible carry propagation hardware. The variety of these adders makes it possible to find the best choice based on the design parameters, e.g., area, delay, number of wiring tracks. Proving the correctness of prefix adders is an important task after their design as they usually have a complex and error-prone structure. It has been experimentally shown that Binary Decision Diagrams (BDDs) are very efficient in the formal verification of adders, including prefix adders. However, it has been never proved theoretically. In this paper, we calculate the computational complexity of proving the correctness of prefix adders using BDDs. Based on these calculations, we show that the formal verification of prefix adders can be done in time polynomial in n, where n is the size of the adder (i.e., the number of bits per input). We also compare the theoretical calculations with the experimental results to clarify the differences between the complexities in theory and practice.**

I. Introduction

An integer adder is one of the most frequent units in many designs and applications. Most of the arithmetic circuits including multipliers and dividers require integer adders in their architectures. Designers have proposed a variety of addition algorithms to satisfy the community demands in terms of area, delay, and the number of wiring tracks. These algorithms usually employ different carry propagation strategies. One of the important groups of adder architectures is prefix adders. Prefix adders are widely used in industry due to their regular structures and efficient design [1].

In prefix adders, addition is expressed as a prefix computation [2]. Using prefix computations makes it possible to have more than one implementation for intermediate structures within the adder, allowing trade-offs between the amount of internal wiring and the fan-out of intermediate nodes. As a result, a large variety of prefix adders can be implemented to satisfy different design goals.

Arithmetic circuits are very error-prone particularly when they are generated automatically. The wrong implementation of an algorithm in an arithmetic circuit generator can result in buggy hardware. It is possible that the bug appears only for some of the circuits' types and sizes, and thus becomes hard to detect. Consequently, an important phase after the design of an arithmetic circuit including a prefix adder is verification. The verification method based on the *Binary Decision Diagram* (BDD) reports very good results when it comes to the verification of integer adders. The core idea of the verification is based on symbolic simulation. In the simulation, an input pattern is applied to a circuit, and the resulting output values are observed to see whether they are the expected values. Symbolic simulation verifies a set of scalar tests in the input space with a single symbolic test. In order to cover all the possible values on each input, symbolic functions are encoded using BDDs. At the end of the simulation, the resulting BDD for each primary output is evaluated. A BDD is a canonical representation; thus, independent of the adder architecture, the outputs' BDDs should be always identical.

Despite the practical success of the BDD-based formal verification method in proving the correctness of different adders, the verification complexity has not been discussed in depth. Therefore, there is a gap in literature where the theoretical proofs for the verification complexities are missing. It leads to several verification problems: 1) unpredictability in performance of the verification method, i.e., it cannot be predicted before actually invoking the verification tool whether it will successfully terminate or run for an indefinite amount of time, 2) unknown scalability of the verification technique, i.e., it is not predictable how much the run-time and the required memory increase when the size of the circuit under verification grows, and 3) It is not possible to compare the performance of the verification method against other techniques and choose the best one.

PolyAdd [3] is the first work that focuses on the complexity of adder verification. The author proves that the complete formal verification process of the three adders (i.e., ripple carry adder, conditional sum adder, and carry look-ahead adder) can be carried out polynomially. This proof becomes possible by calculating the BDD sizes for each internal signal. PolyAdd focuses on only three traditional adder architectures, and it does not ensure the polynomial verification of other adder structures such as prefix adders. Moreover, the calculations for the exact order of the complexity are absent in the paper. Authors of [4] extend PolyAdd by extracting the exact verification complexity bounds for the conditional sum adder.

In this paper, we calculate the computational complexity of proving the correctness of three prefix adders: serial prefix, Ladner-Fischer, and Kogge-Stone adder. Moreover, we prove that the computational complexity of verifying the prefix adders is always polynomial with respect to the number of bits per input. This new achievement confirms the good scalability of the BDD-based verification method in proving the correctness of prefix adders, which had been only shown in practice before. Finally, we compare the theoretical calculations with the experimental results to clarify the differences between the complexities in theory and practice. This paper is the first step to fill the long-lasting gap in the BDD-based verification of adders by providing a careful exposition of the computational steps to obtain the polynomial bounds.

II. Preliminaries

In this section, first, an overview of the prefix adders is given. Then, the BDD representation is reviewed.

A. Prefix Adders

A simple ripple carry adder is very area-efficient; however, it has a huge delay due to the long carry chains. Several solutions have been proposed to solve the problem of carry propagation

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

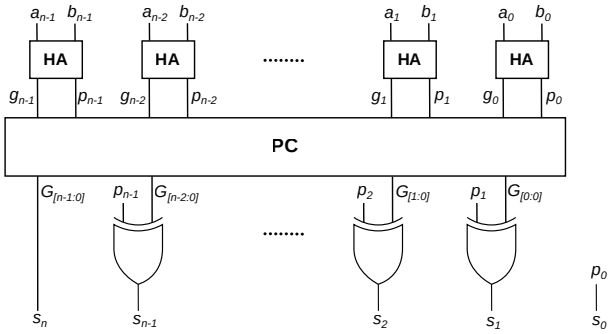


Fig. 1. The general structure of a prefix adder

and provide flexibility in computing carries. Among these solutions, prefix algorithms gained high popularity as they can be used to generate a wide variety of adders aiming for different design goals.

We explain the mathematical foundation of the prefix adders in detail. Let $A = a_{n-1}a_{n-2}\dots a_0$ and $B = b_{n-1}b_{n-2}\dots b_0$ be n-bit binary numbers. Their sum can be computed as $S = A + B = s_n s_{n-1}\dots s_0$ which has $n+1$ bits. For each position $0 \leqslant i \leqslant n-1$, we compute a *generate* signal g_i and a *propagate* signal p_i:

$$\begin{aligned} g_i &= a_i \wedge b_i, \\ p_i &= a_i \oplus b_i \end{aligned} \tag{1}$$

where $\wedge$ and $\oplus$ are binary AND and XOR, respectively. Please note that g_i and p_i signals can be generated from primary inputs using *Half-adders* (HAs). Then, the carry bits are computed recursively using generate and propagate signals:

$$c_{i+1} = g_i \vee (p_i \wedge c_i) \tag{2}$$

By having carry bits in hand, we can compute the final sum bits:

$$s_i = \begin{cases} c_i \oplus p_i & \text{if } 0 \leqslant i \leqslant n-1 \\ c_i & \text{if } i = n \end{cases} \tag{3}$$

For two pairs (g_i, p_i) and (g_j, p_j) of generate and propagate signals, the binary *prefix operator* is defined as:

$$\begin{pmatrix} g_i \\ p_i \end{pmatrix} \bullet \begin{pmatrix} g_j \\ p_j \end{pmatrix} = \begin{pmatrix} g_i \vee (p_i \wedge g_j) \\ p_i \wedge p_j \end{pmatrix} \tag{4}$$

The prefix operator can be used to compute the carry bit c_{i+1}:

$$\begin{pmatrix} c_{i+1} \\ p_i \wedge p_{i-1} \wedge \dots \wedge p_0 \end{pmatrix} = \begin{pmatrix} g_i \\ p_i \end{pmatrix} \bullet \begin{pmatrix} g_{i-1} \\ p_{i-1} \end{pmatrix} \bullet \dots \bullet \begin{pmatrix} g_0 \\ p_0 \end{pmatrix} \tag{5}$$

The most important advantage of the prefix operator is its associativity. Thus, the carry bits can be computed in many different ways. We define $G_{[i:j]}$ and $P_{[i:j]}$ to show the result of applying prefix operator to $i-j$ consecutive generate and propagate signals, respectively:

$$\begin{pmatrix} G_{[i:j]} \\ P_{[i:j]} \end{pmatrix} = \begin{pmatrix} g_i \\ p_i \end{pmatrix} \bullet \begin{pmatrix} g_{i-1} \\ p_{i-1} \end{pmatrix} \bullet \dots \bullet \begin{pmatrix} g_j \\ p_j \end{pmatrix} \tag{6}$$

Based on the definition of $G_{[i:j]}$ and $P_{[i:j]}$, we can derive the following equations:

$$\begin{pmatrix} G_{[i:j]} \\ P_{[i:j]} \end{pmatrix} = \begin{pmatrix} G_{[i:k]} \\ P_{[i:k]} \end{pmatrix} \bullet \begin{pmatrix} G_{[k-1:j]} \\ P_{[k-1:j]} \end{pmatrix} \tag{7}$$

$$c_{i+1} = G_{[i:0]} \tag{8}$$

$G_{[i:j]}$ and $P_{[i:j]}$ are used in sophisticated parallel prefix adders to compute the carry bits.

Fig. 1 shows the general structure of a prefix adder [2], [5]. First, the propagate (p_i) and generate (g_i) signals are computed from the primary inputs using HAs. Then, the prefix computations are performed in the PC block to generate the carry bits (i.e., $G_{[i:0]}$). As we mentioned, the associativity of prefix operator (see Eq. (5), Eq. (6), and Eq. (7)) makes it possible to compute the carry bits in many different ways. Several prefix algorithms have been proposed to compute the carry bits. Each algorithm prioritizes one of the important design parameters (e.g., area, delay, or the number of wiring tracks) or tries to make a trade-off between them. As a result, the PC block is a tree of prefix operators connected based on a prefix algorithm, e.g., Ladner-Fischer or Kogge-Stone. Finally, the sum bits (s_i) are computed in the final stage using XOR gates based on Eq. (3). We now explain the PC block of the three prefix adders in detail:

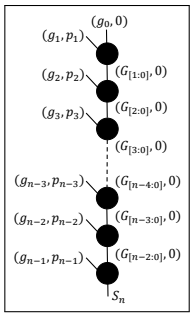


Fig. 2. Serial prefix adder

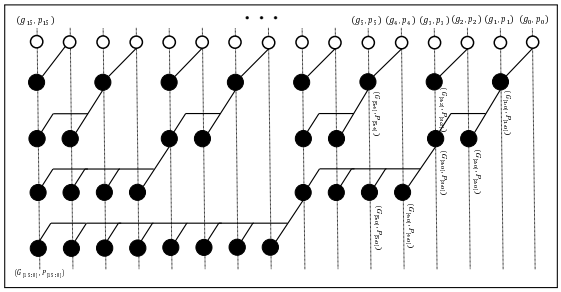


Fig. 3. Ladner-Fischer adder

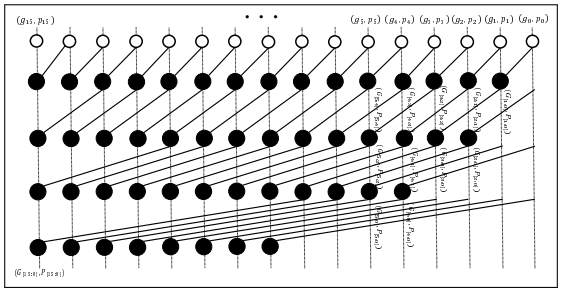


Fig. 4. Kogge-Stone adder

Serial Prefix Adder: Fig. 2 represents the PC block of a serial prefix adder. Prefix operators are arranged in a single column. The serial prefix algorithm needs a minimal number of binary operations, but it is inherently slow.

Ladner-Fischer Adder: Fig. 3 depicts the PC block of a 16-bit Ladner-Fischer adder. In this architecture, intermediate signals are computed by a minimal tree structure and distributed in parallel to all higher bit positions. This structure leads to a high fan-out of some prefix operators, but on the other hand, it results in the smallest possible delay and very few wiring tracks.

Kogge-Stone Adder: Fig. 4 shows the structure of a 16-bit Kogge-Stone adder. This architecture has minimal depth

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** ▷ General case
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

(like the Ladner-Fischer adder) as well as bounded fan-outs (maximum of 2 fan-outs for each node). However, using a large number of independent and parallel tree structures has led to a massively increased number of prefix operators and interconnections.

B. Binary Decision Diagrams

Definition 1. *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 2. *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 3. *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 logic operations in BDDs:

$$ITE(f, g, h) = (f \wedge g) \vee (\bar{f} \wedge h) \tag{9}$$

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

$$\begin{aligned} f \wedge g &= ITE(f, g, 0), \qquad & f \vee g &= ITE(f, 1, g), \\ f \oplus g &= ITE(f, \overline{g}, g), & \bar{f} &= ITE(f, 0, 1). \end{aligned} \tag{10}$$

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

$$ITE(f, g, h) = ITE(x_i, ITE(f_{x_i}, g_{x_i}, h_{x_i}), ITE(f_{\overline{x}_i}, g_{\overline{x}_i}, h_{\overline{x}_i})) \tag{11}$$

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).

The algorithm for calculating ITE operations is presented in Algorithm 1. The result is computed recursively based on Eq. (11) 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 subdiagrams of f, g and h. The algorithm employs two major data structures: a *Unique Table* to guarantee the canonicity of the BDDs (see Line 9), and a *Computed Table* to store results of previous computations and avoid repetition (see Line 10). The number of subdiagrams in a BDD is equivalent to the number of nodes. For each of the three arguments, the sub-routine is called at most once. Assuming that the search in the *Unique Table* is performed at a constant time, the computational complexity of the ITE algorithm, even in the worst-case, does not exceed $O(|f| \cdot |g| \cdot |h|)$, where $|f|$, $|g|$ and $|h|$ denote the size of the BDDs in terms of the number of nodes [6].

In order to formally verify an adder, we need to have the BDD representation of the outputs. Symbolic simulation helps us to obtain the BDD for each primary output. In a simulation, an input pattern is applied to a circuit, and the resulting output values are observed to see whether they match the expected values. On the other hand, symbolic simulation verifies a set of scalar tests (which usually covers 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 an adder.

III. Complexity of Verifying Prefix Adders

In this section, we first calculate the complexity of the symbolic simulation for the first and third stages of a prefix adder (Fig. 1). Then, we derive the complexity of a prefix operation. Finally, we use the obtained results to calculate the computational complexity of verifying different prefix adders.

A. First and Third Stage Complexity

Prefix adders have different architectures in their PC block. However, the first stage (i.e., HAs) and third stage (i.e., XOR gates) are identical in all types of prefix adders (see Fig. 1). Thus, the computational complexities of these two stages in verification are exactly the same between prefix adders.

A HA consists of an AND($\wedge$) and an XOR($\oplus$) operators; therefore, it can be translated into two ITE operations:

$$\begin{aligned} g_i &= a_i \wedge b_i = ITE(a_i, b_i, 0), \\ p_i &= a_i \oplus b_i = ITE(a_i, \overline{b}_i, b_i) \end{aligned} \tag{12}$$

The ITE operations are computed by Algorithm 1 to get the BDDs for the g_i and p_i signals. Assuming that f, g and h are the input arguments of an ITE operator, the computational complexity is computed as $|f| \cdot |g| \cdot |h|$. As a result, the complexity of computing g_i and p_i is as follows:

$$\begin{aligned} complexity(g_i) &= |a_i| \cdot |b_i| = 9, \\ complexity(p_i) &= |a_i| \cdot |\overline{b}_i| \cdot |b_i| = 27 \end{aligned} \tag{13}$$

Since there are n HAs in an n-bit prefix adder, the computational complexity of the first stage is:

$$complexity_{[stage1]} = 36 \cdot n \tag{14}$$

In the final stage of a prefix adder, the XOR operations are required to generate the sum bits. These operations can be translated into ITE operations:

$$s_i = G_{[i-1:0]} \oplus p_i = ITE(G_{[i-1:0]}, \overline{p}_i, p_i) \tag{15}$$

The complexity of the ITE operation is calculated as follows:

$$complexity(s_i) = |G_{[i-1:0]}| \cdot |\overline{p}_i| \cdot |p_i| \tag{16}$$

It has been proved in [7] that the BDD size of the i^{th} carry bit ($G_{[i-1:0]}$) is bounded by $3(i+1)$. We substitute the size of $G_{[i-1:0]}$, $\overline{p}_i$, and p_i in Eq. (16) to get:

$$complexity(s_i) = (3 \cdot (i+1)) \cdot 5 \cdot 5 = 75 \cdot (i+1) \tag{17}$$

By adding up the complexity of all XOR operations, we get the computational complexity of the third stage:

$$complexity_{[stage3]} = \sum_{i=1}^{n-1} 75 \cdot (i+1) = \frac{75}{2} n^2 + \frac{75}{2} n - 75 \tag{18}$$

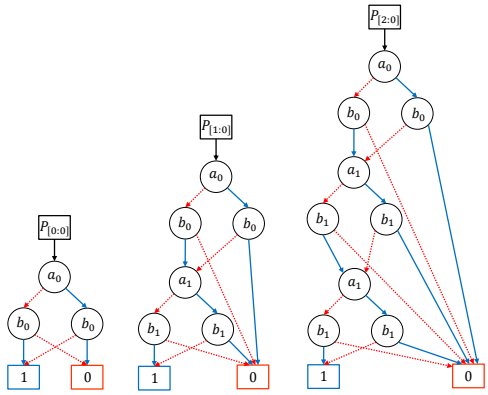


Fig. 5. The BDDs for $P_{[0:0]}$, $P_{[1:0]}$, and $P_{[2:0]}$

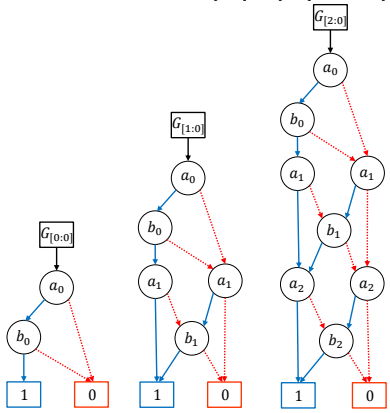


Fig. 6. The BDDs for $G_{[0:0]}$, $G_{[1:0]}$, and $G_{[2:0]}$

B. Prefix Operation Complexity

Based on the definition of a prefix operator (see Eq. (4) in Section II), the prefix operation between two pairs $(G,P)_{[i:k]}$ and $(G,P)_{[k-1:j]}$ (see Eq. (7)) is given by the two Boolean functions:

$$\begin{aligned} G_{[i:j]} &= G_{[i:k]} \vee (P_{[i:k]} \wedge G_{[k-1:j]}), \\ P_{[i:j]} &= P_{[i:k]} \wedge P_{[k-1:j]} \end{aligned} \tag{19}$$

In order to obtain the BDDs for the Boolean functions, they are first converted into ITE operations:

$$\begin{aligned} G_{[i:j]} &= ITE(G_{[i:k]}, 1, P_{[i:k]} \wedge G_{[k-1:j]}) \\ &= ITE(G_{[i:k]}, 1, ITE(P_{[i:k]}, G_{[k-1:j]}, 0)), \\ P_{[i:j]} &= ITE(P_{[i:k]}, P_{[k-1:j]}, 0) \end{aligned} \tag{20}$$

Thus, the complexity of computing $G_{[i:j]}$ and $P_{[i:j]}$ is as follows:

$$\begin{aligned} Complexity(G_{[i:j]}) &= |G_{[i:k]}| \cdot |P_{[i:k]}| \cdot |G_{[k-1:j]}|, \\ Complexity(P_{[i:j]}) &= |P_{[i:k]}| \cdot |P_{[k-1:j]}| \end{aligned} \tag{21}$$

We need to calculate the size of the BDDs and substitute them in Eq. (21) to get the complexities based on the i, j, and k variables.

According to Eq. (4) and Eq. (6), $P_{[m:n]}$ is computed by preforming AND operations between $m-n$ propagate signals:

$$P_{[m:n]} = p_m \wedge p_{m-1} \wedge \cdots \wedge p_n \tag{22}$$

The propagate signal p_t can be substituted by $a_t \oplus b_t$ to get the Boolean function for $P_{[m:n]}$ based on the primary inputs:

$$P_{[m:n]} = (a_m \oplus b_m) \wedge (a_{m-1} \oplus b_{m-1}) \wedge \cdots \wedge (a_n \oplus b_n) \tag{23}$$

The size of the BDD for $P_{[m:n]}$ depends on the number of XOR operations. Fig. 5 presents the BDDs for $P_{[0:0]}$, $P_{[1:0]}$ and $P_{[2:0]}$ in which there are one, two and three XOR operations, respectively. There are in total $m-n+1$ XOR operators in each function, and each XOR operator adds three nodes to the BDD. Thus, considering the two terminal nodes, the size of the BDD for $P_{[m:n]}$ is obtained:

$$|P_{[m:n]}| = 3 \cdot (m-n+1) + 2 = 3 \cdot (m-n) + 5 \tag{24}$$

According to Eq. (4) and Eq. (7), $G_{[i:j]}$ can be computed as follows:

$$G_{[m:n]} = g_m \vee (p_m \wedge G_{[m-1:n]}) \tag{25}$$

In order to get the function based on the primary inputs, we substitute g_i and p_i with $a_i \wedge b_i$ and $a_i \oplus b_i$, respectively:

$$G_{[m:n]} = (a_m \wedge b_m) \vee ((a_m \oplus b_m) \wedge G_{[m-1:n]}) \tag{26}$$

Fig. 6 shows the BDDs for $G_{[0:0]}$, $G_{[1:0]}$ and $G_{[2:0]}$. The initial BDD size for $G_{[0:0]} = g_0$ is equal to 4, and performing prefix operation with g_1 increases the size of the BDD by 3 nodes. The increase in the size of the BDD occurs again after the operation with g_2, g_3, and so forth. Therefore, the BDD size for $G_{[i:j]}$ is calculated as follows:

$$|G_{[m:n]}| = 3 \cdot (m-n) + 4 \tag{27}$$

By replacing the size of $G_{[i:k]}$, $G_{[k-1:j]}$, $P_{[i:k]}$, and $P_{[k-1:j]}$ BDDs in Eq. (21) based on Eq. (24) and Eq. (27), the complexity of computing $G_{[i:j]}$ and $P_{[i:j]}$ is obtained:

$$\begin{aligned} &Complexity(G_{[i:j]}) = \\ &\quad (3 \cdot (i-k)+4) \cdot (3 \cdot (i-k)+5) \cdot (3 \cdot (k-j)+1), \\ &Complexity(P_{[i:j]}) = \\ &\quad (3 \cdot (i-k)+5) \cdot (3 \cdot (k-j)+2) \end{aligned} \tag{28}$$

The calculated complexities only depend on $i-k$ and $k-j$. Thus, we can represent them as two functions $CG(i-k, k-j)$ and $CP(i-k, k-j)$ with only two input arguments.

A prefix operation contains the computations for both $G_{[i:j]}$ and $P_{[i:j]}$. Thus, the overall complexity can be calculated:

$$C(i-k, k-j) = CG(i-k, k-j) + CP(i-k, k-j) \tag{29}$$

We use the $C(i-k, k-j)$ function to calculate the complexity of the three prefix adders.

C. Verification Complexity of a Serial Prefix Adder

In an n-bit serial prefix adder, the carry bits (i.e., $G_{[t:0]}$) are computed subsequently using prefix operators (see Fig. 2). The inputs of the t^{th} prefix operator are 1) the pair of generate (g_{t+1} or $G_{[t+1:t+1]}$) and propagate (p_{t+1} or $P_{[t+1:t+1]}$) signals, and 2) the carry bit ($G_{[t:0]}$) generated by the previous prefix operator. As a result, the t^{th} prefix operator always performs the following operation:

$$\begin{pmatrix} G_{[t+1:0]} \\ 0 \end{pmatrix} = \begin{pmatrix} G_{[t+1:t+1]} \\ P_{[t+1:t+1]} \end{pmatrix} \bullet \begin{pmatrix} G_{[t:0]} \\ 0 \end{pmatrix}, \tag{30}$$

where the second signal in the output pair is always equal to zero, and therefore it requires no computation. Thus, CP is equal to zero in all calculations.

The computational complexity of t^{th} prefix operation is calculated based on Eq. (28) and Eq. (29) as follows:

$$\begin{aligned} C(t+1-t-1, t+1-0) &= C(0, t+1) = CG(0, t+1) \\ &= 60t + 80 \end{aligned} \tag{31}$$

The computational complexity of the PC block is calculated by adding up the complexity of each individual prefix operation.

$$complexity_{[PC]} = \sum_{t=1}^{n} (60t + 80) = 30n^2 + 110n \tag{32}$$

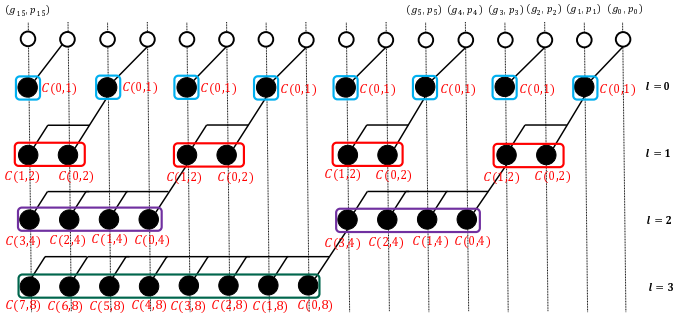


Fig. 7. The complexity calculations for a 16-bit Ladner-Fischer adder

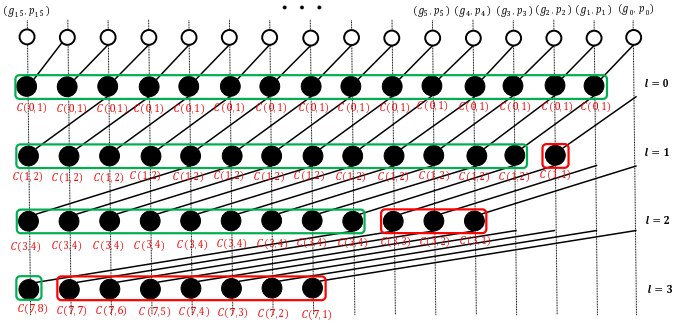


Fig. 8. The complexity calculations for a 16-bit Kogge-Stone adder

Finally, the overall computational complexity of a serial prefix adder is achieved by adding up the complexity of the three stages in Eq. (14), Eq. (18), and Eq. (32). We can conclude that the order of the verification complexity is $\mathcal{O}(n^2)$. As a result, proving the correctness of a serial prefix adder has quadratic time complexity.

D. Verification Complexity of a Ladner-Fischer adder

The Lander-Fischer adder is a parallel prefix adder in which the prefix operators are connected in a tree structure to generate the carry bits in parallel. Fig. 3 shows the intermediate pairs $(G, P)_{[m:n]}$ generated by the prefix operators. By knowing the inputs of each prefix operator (i.e., $(G, P)_{[i:k]}$ and $(G, P)_{[k-1:j]}$), the computational complexity can be calculated using function $C(i-k, k-j)$ (see Eq. (28) and Eq. (29)). Fig. 7 presents the complexity of the prefix operation for each node of the circuit by the C function.

For an n-bit Ladner-Fischer adder, the depth (i.e., number of rows) and the number of nodes in each row are calculated as follows:

$$\begin{aligned} depth &= \log_2(n), \\ nodes\_in\_row &= \frac{n}{2}, \end{aligned} \tag{33}$$

where the equations are exact for all word length being a power of 2 (i.e., $n = 2^m$) [2].

We group the nodes with the same fan-ins in each row of a Ladner-Fischer adder, e.g., the second row is divided into four groups in which the nodes have the same fan-ins (see red boxes in Fig. 7). The number of groups in each row and the number of nodes in each group depend on the row number l, and are calculated as follows:

$$\begin{aligned} numder\_of\_groups &= 2^{(\log_2 \frac{n}{2} - l)}, \\ nodes\_in\_group &= 2^l \end{aligned} \tag{34}$$

As an example, the second row ($l = 1$) has $2^{(\log_2 \frac{16}{2} - 1)} = 4$ groups, and there are $2^1 = 2$ nodes in each group.

The second input argument of the C function (i.e., $k-j$) is identical for the nodes in a group and equals 2^l. On the other hand, the first argument (i.e., $i-k$) is equal to h for the h^{th} node in the group. Consequently, the groups in each row have the same accumulative complexity, which can be obtained by adding up the complexity of the nodes:

$$group\_complexity = \sum_{h=0}^{2^l-1} C(h, 2^l) \tag{35}$$

The computational complexity of a whole row is obtained with respect to the complexity of the groups and the number of groups in a row:

$$row\_complexity = 2^{(\log_2 \frac{n}{2} - l)} \times \sum_{h=0}^{2^l-1} C(h, 2^l) \tag{36}$$

The computational complexity of the PC block is obtained by adding up the complexity of all rows:

$$\begin{aligned} complexity_{[PC]} &= \sum_{l=0}^{\log_2(n)-1} \left(2^{(\log_2 \frac{n}{2} - l)} \times \sum_{h=0}^{2^l-1} C(h, 2^l) \right) \\ &= \frac{9}{14}n^4 + \frac{23}{4}n^3 + \frac{93}{4}n^2 + \frac{15}{2}n\log_2 n - \frac{415}{14}n \end{aligned} \tag{37}$$

Finally, the overall computational complexity of a Ladner-Fischer adder is calculated by adding up the complexity of the three stages in Eq. (14), Eq. (18), and Eq. (37). Based on the calculated complexity, we can observe that the order of the verification complexity is $\mathcal{O}(n^4)$. Therefore, proving the correctness of a Ladner-Fischer adder using BDDs has quartic time complexity.

E. Verification Complexity of a Kogge-Stone adder

The Kogge-Stone adder is another parallel prefix adder with a parallel tree of prefix operators (see Fig. 4). The computational complexity of each prefix operator is shown in Fig. 8 as a C function. Note that if the inputs of a prefix operator are $(G_{[i:k]}, P_{[i:k]})$ and $(G_{[k-1:j]}, P_{[k-1:j]})$, the complexity can be calculated by $C(i-k, k-j)$.

For an n-bit Kogge-Stone adder, the depth (i.e., number of rows) and the number of nodes in each row are:

$$\begin{aligned} depth &= \log_2(n), \\ nodes\_in\_row &= n - 2^l \end{aligned} \tag{38}$$

where l is the row number. Please note that the equations are exact for all word lengths being a power of 2 (i.e., $n = 2^m$) [2].

We divide the nodes in each row into two groups based on the input values of the C functions. In the first group (green boxes in Fig. 8), the input values of the C functions are identical, i.e., $C(2^l - 1, 2^l)$. In the second group (red boxes in Fig. 8), the first input values are exactly the same and equal $2^l - 1$. However, the second value is equal to $h + 1$ for the h^{th} node in the group.

The number of nodes in the first group ($group1$) and the second group ($group2$) are as follows:

$$\begin{aligned} nodes\_in\_group1 &= n - 2^{l+1} + 1, \\ nodes\_in\_group2 &= 2^l - 1 \end{aligned} \tag{39}$$

The computational complexity of each group is obtained by

TABLE I
RUN-TIME OF VERIFYING ADDERS (SECONDS)

Size	Benchmarks		
	serial prefix	Ladner-Fischer	Kogge-Stone
1024	1.28	1.64	1.84
2048	6.37	7.56	8.37
3072	15.24	17.94	21.60
4096	27.21	33.59	39.01
5120	43.05	49.85	69.89
6144	67.87	78.07	104.47
7168	97.36	114.06	142.42
8192	129.78	153.67	177.43
9216	164.53	184.33	234.78
10240	200.45	241.49	315.52

adding up the complexity of the inside nodes:

$$group1\_complexity = (n - 2^{l+1} + 1) \times C(2^l - 1, 2^l),$$
$$group2\_complexity = \sum_{h=0}^{2^l-2} C(2^l - 1, h + 1) \quad (40)$$

We can add the complexity of the first and second groups to get the computational complexity of a row. The computational complexity of the PC block is obtained by adding up the complexity of all rows:

$$complexity_{[PC]} = \sum_{l=0}^{\log_2(n)-1} \left((n - 2^{l+1} + 1) \times C(2^l - 1, 2^l) + \sum_{h=0}^{2^l-2} C(2^l - 1, h + 1) \right) = \frac{81}{70}n^4 + \frac{111}{14}n^3 + 22n^2 + 6n\log_2 n - \frac{321}{7}n + \frac{517}{35} \quad (41)$$

By adding up the complexity of the three stages in Eq. (14), Eq. (18), and Eq. (41), the overall complexity is obtained. After calculating the computational complexity, we can conclude that the order of the BDD-based verification complexity is $\mathcal{O}(n^4)$. Therefore, proving the correctness of a Kogge-Stone adder has quartic time complexity.

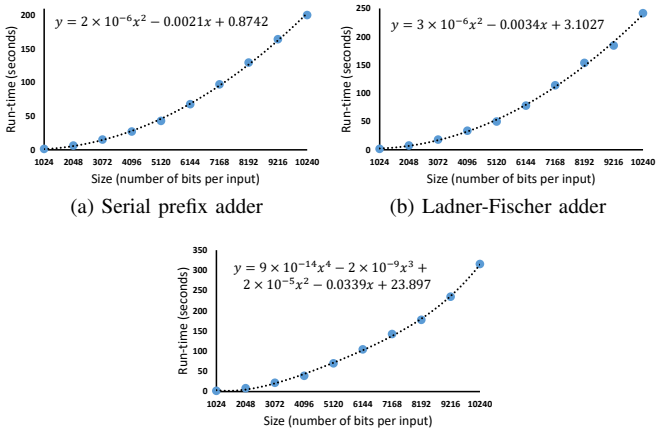
IV. EXPERIMENTAL RESULTS

We have implemented the BDD-based verifier in C++. The tool takes advantage of the symbolic simulation to obtain the BDDs for the primary outputs. Then, the BDDs are evaluated to see whether they match the BDDs for an adder. In order to handle the BDD operations, we used the CUDD library [8]. The benchmarks for the three prefix adders are generated using GenMul [9]. All experiments are performed on an Intel(R) Core(TM) i7-8565U with 1.80 GHz and 24 GByte of memory.

Table I reports the verification times for adders. The first column **Size** denotes the size of the adder based on the inputs' bit-width. The run-time (in seconds) of the BDD-based verification method is reported in the second column **Benchmarks** for the three prefix adders.

It is evident in Table I that the BDD-based verification reports very good results for prefix adders. A Kogge-Stone adder with 10240 bits per input, which consists of more than 400K gates, can be verified in less than 6 minutes. Thus, the experimental results for the three prefix adders confirm the scalability of the BDD-based verification method.

In order to check the correctness of the complexity bounds obtained in Section III, we first show the results of Table I as three graphs in Fig. 9. Then, we fit a curve to the points with an acceptable error and evaluate the curve function. We can fit a curve with the order of 2 to the verification run-times of a serial prefix adder in Fig. 9a. It confirms the calculated complexity bound in Eq. (32). It is also possible to fit a polynomial with the order of 2 to the verification



(a) Serial prefix adder

(b) Ladner-Fischer adder

(c) Kogge-Stone adder

Fig. 9. Run-time graphs of the prefix adders

run-times of a Ladner-Fischer adder (see Fig. 9b). It has a smaller order of complexity than our theoretical calculations in Eq. (37). However, the theoretical complexity calculations are always based on the worst-case scenarios that might never happen in practice. Finally, a polynomial curve with the order of 4 is fitted to the verification run-times of a Kogge-Stone adder. It confirms our theoretical calculation for obtaining the computational complexity of a Kogge-Stone adder in Eq. (41).

V. CONCLUSION

In this paper, we calculated the computational complexity of the BDD-based verification for the three prefix adders, i.e., serial prefix adder, Ladner-Fischer adder, and Kogge-Stone adder. Based on the calculations, we proved that verifying these adders is possible in polynomial time. The experimental results confirm the correctness of the complexity bounds obtained in our theoretical calculations.

In our future research, we focus on the verification complexity of the other prefix adders, e.g., Brent-Kung adder and Han-Carlson adder. We also investigate the complexity of the other existing verification techniques such as *BMD [10], [11] and SCA [12], [13], [14] when it comes to proving the correctness of different arithmetic circuits.

REFERENCES

[1] S. Knowles, "A family of adders," in *ARITH*, 2001, pp. 277–284.
[2] R. Zimmermann, "Binary adder architectures for cell-based VLSI and their synthesis," Ph.D. dissertation, Swiss Federal Institute of Technology, 1997.
[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.
[5] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. A. K. Peters, Ltd., 2001.
[6] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *DAC*, 1990, pp. 40–45.
[7] I. Wegener, *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.
[8] F. Somenzi, "CUDD: CU decision diagram package release 2.7.0," available at https://github.com/ivmai/cudd, 2018.
[9] 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.
[10] R. Drechsler, B. Becker, and S. Ruppertz, "The K*BMD: A verification data structure," *IEEE Design & Test of Computers*, vol. 14, no. 2, pp. 51–59, 1997.
[11] M. Keim, R. Drechsler, B. Becker, M. Martin, and P. Molitor, "Polynomial formal verification of multipliers," *Formal Meth. in Sys. Des.*, vol. 22, no. 1, pp. 39–58, 2003.
[12] 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.
[13] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.
[14] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019, pp. 28–36.