Mathematical and Computer Modelling 52 (2010) 191–199

Contents lists available at ScienceDirect

Mathematical and Computer Modelling

journal homepage: www.elsevier.com/locate/mcm

Formal proof of prefix adders

Feng Liu [a,*], Qingping Tan [a], Gang Chen [b]

[a] National Lab of Parallel Distributed Processing, Hunan, China

[b] Lingcore Lab, Portland, OR, USA

ARTICLE INFO

Article history:
Received 12 July 2009
Received in revised form 7 February 2010
Accepted 9 February 2010

Keywords:
Prefix adders
Computer arithmetic
Semi-group
VLSI

ABSTRACT

The paper presents an algebraic analysis for the correctness of prefix-based adders. In contrast to using higher-order functions and rewriting systems previously, we harness first-order recursive equations for correctness proof. A new carry operator is defined in terms of a semi-group with the set of binary bits. Both sequential and parallel addition algorithms are formalized and analyzed. The formal analysis on some special prefix adder circuits demonstrates the effectiveness of our algebraic approach. This study lays an underpinning for further understanding on computer arithmetic systems.

1. Introduction

Binary addition is the most fundamental and frequently used operation. It has the largest impact on the overall computer performance [1]. With the shrinking feature size and the growing density, the costs of VLSI circuit manufacturing are soaring, so does the cost of circuit errors. Therefore, the logic correctness of binary addition algorithm becomes extremely important. In the area of computer arithmetic, a great number of addition algorithms have been proposed. At the meantime, the linear time adders are too slow for modern systems, the speed of parallel carry propagation adders is required. Classic high-speed adders include Carry Lookahead Adder (CLA), parallel prefix adders and so on.

It has been well known that the Carry Lookahead Adder plays a key role in the development of different generations of adders, such as parallel prefix adders. These parallel adders have been extensively studied in the past few years [2–6]. Some inventors of parallel prefix adder algorithms have presented correctness proofs for their algorithms, for example, Kogge and Stone [4]. Brent and Kung proved a few useful lemmas in their famous paper [5]. However, most adder algorithms are based on a few elegant properties, which are often accepted by intuition. Formal analysis on their correctness is limited, or less well known. The details of their proofs are either incomplete or hard to find. In this paper, we collect existing formal results and present a set of new properties concerning prefix adders. Based on the traditional notions such as Propagated Carry and Generated Carry, we present a detailed proof for generic prefix adder. In our approach, the representation of the algorithm is described in the form of first-order recursive equations, which are widely adopted in computer arithmetic community. Therefore, our proof is direct and more accessible to VLSI designers. In particular, we introduce a new definition of fundamental carry operator, which is given in terms of Boolean operations on bit pairs. Carry propagation functions which can be represented as bit pairs are composed by the fundamental carry operator. This makes it easy to analyze the essential algebraical properties of fundamental carry operator at a deeper level. This insight is central to the operation of prefix adders. We also prove that the set of bit pairs and fundamental carry operator constitute a monoid. Using this algebraic structure, we formalize a special parallel prefix adder Kogge–Stone tree and prove its correctness in a new way. It is our belief that

* Corresponding author.
E-mail address: liuprayer@gmail.com (F. Liu).

0895-7177/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.mcm.2010.02.008

these proofs would be helpful for people to get a better understanding about the nature of adder algorithms. Another aim is to make it convenient for further formal investigations.

The rest of this paper is structured as follows. Section 2 reviews some related works. In Section 3, preliminaries and some existing formal results are introduced. In Section 4, we present the correctness proofs of liner time prefix adders. They are the base of advanced proofs of parallel prefix adders. In Section 5, we develop some useful theorems and given the proof framework of parallel prefix adders. Section 6 concludes this paper.

2. Related work

There were some interesting works related to the correctness proofs of adder algorithms. In paper [7], high-order functional combinators are employed to represent Carry Lookahead Adder and the correctness is established by algebraic transformations. D. Kapur and M. Subramaniam described adders' algorithms using the language of RRL and verified the correctness by term rewriting in a rewrite-rule based theorem prover [8]. Mary Sheeran recently made a profound formal investigation into parallel prefix computation using the Haskell language [9]. R. Hinze employed the Haskell as the meta language and introduced an algebra of scans which can be used in formal analysis of parallel prefix circuits [10]. These works used many jargons of formal verification specialists. Gang Chen et al. [11] proposed a concise presentation for the proof of correctness and properties of integer adders. In particular, these proofs are based on first-order recursive equations, which, we believe, is more accessible to VLSI designers than functional and rewriting style proofs. But it doesn't present any specific parallel prefix adder which must use some sort of network to perform the group carry computations in parallel. This paper was primarily inspired by the work [11]and is a fairly natural extension of it. We provide machinery that can be applied to any adder based on group carry computations. The new contributions in this paper include the new definition of fundamental carry operator, analysis of its algebraic properties, presentation of the correctness proof of a typical parallel prefix adder (Kogge–Stone adder) using the algebraic structures we built and so on.

The fundamental carry operator and its associativity is discovered by Brent and Kung [5]. In this paper, we introduced an new definition of fundamental carry operator. We proved that the new fundamental carry operator and the set of bit pairs constitute a monoid. S. Lakshmivarahan and S.K. Dhall also found that the computation of the lookahead adder's carry bit can be reformulated as a semi-group product [12]. But they didn't give the correctness proof. In this paper, we build the monoid in different way.

3. Preliminaries

We need some basic notions to introduce our formulations.

In this paper, the symbols 0 and 1 denote Boolean False and True, or digital number Zero and One; the symbol $\wedge$ denotes the Boolean AND; $\vee$ denotes the Boolean OR; $\oplus$ denotes the Boolean Exclusive OR.

In the following, we describe some useful algebraic structures: semi-group and monoid.

Definition 3.1 (*Semi-group [13]*)**.** Let A be a nonempty set and $\diamond$ a binary operation over A. $(A, \diamond)$ is a semi-group if it satisfies the following conditions:

1. (A is closed under $\diamond$) $a, b \in A$, then $a \diamond b \in A$;
2. (Associativity) For any $a, b, c \in A$, then $(a \diamond b) \diamond c = a \diamond (b \diamond c)$.

Due to the associativity, we may safely write: $a \diamond b \diamond c = (a \diamond b) \diamond c = a \diamond (b \diamond c)$.

Definition 3.2 (*Monoid [13]*)**.** Let $(A, \diamond)$ be a semi-group. $(A, \diamond)$ is a monoid if there exists an element e of A, let d be any element of A, then $e \diamond d = d = d \diamond e$. e is called an identity element of $(A, \diamond)$.

Semi-group has some various properties of interest, we list the idempotent below. It is helpful to analyze the algorithms of adders. Interested readers can refer to [13] for more details about semi-group.

Property 3.3 (*Idempotent*)**.** *Let a be an element of a semi-group $(A, \diamond)$. If $a \diamond a = a^2 = a$, then a is called an idempotent.*

Clearly, for a monoid, the identity is always an idempotent.

The binary number representation system is fundamental to describe the binary addition algorithms. Typically, a binary number of length n ($n \geq 0$) is an ordered sequence of binary bits where each bit can assume one of the values 0 or 1. We use $x = (x_{n-1}x_{n-2} \cdots x_0)$ to denote a N-bit binary number; x_i is the binary bits at position i, where $0 \leq i \leq n-1$. The integer value of $x = (x_{n-1}x_{n-2} \cdots x_0)$ is defined as:

$$Int(x) = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2 + x_0 = \sum_{i=0}^{n-1} x_i 2^i.$$

For two N-bits binary numbers $x = (x_{n-1}x_{n-2} \cdots x_0)$ and $y = (y_{n-1}y_{n-2} \cdots y_0)$, let $c = \{c_n, c_{n-1}, \ldots, c_0\}$ be the set of carries where c_0 is the initial incoming carry, c_i denotes the carry from the bit position $i-1$, $0 < i \leq n-1$; the conventional

algorithm for binary addition which computes the sum $s = (c_n s_{n-1} s_{n-2} \cdots s_0)$ is well known and can be formalized as the following equations:

$$\begin{aligned} s_i &= x_i \oplus y_i \oplus c_i \\ c_i &= (x_i \wedge y_i) \vee (x_i \wedge c_{i-1}) \vee (y_i \wedge c_{i-1}), \quad \text{for } i = 0, \ldots, n-1. \end{aligned} \tag{1}$$

The correctness of these equations can be expressed by

$$\sum_{i=0}^{n-1} x_i 2^i + \sum_{i=0}^{n-1} y_i 2^i + c_0 = \sum_{i=0}^{n-1} s_i 2^i + c_n 2^n. \tag{2}$$

The correction of (1) can be intuitively understood. Here, we give the complete proof. This proof depends on properties relating to binary logical operation and arithmetic computation as shown in Lemmas 3.4 and 3.5. Lemmas 3.4 and 3.5 are developed in paper [11].

Lemma 3.4 (*Bit Addition [11]*). *Assume a, b are two bits, then $a + b = 2(a \wedge b) + a \oplus b$.*

Lemma 3.5. *Assume a, b, c are three bits, then $a + (b \wedge c) = a \vee (b \wedge c)$.*

The idea of our proof is similar to paper [11], but some parts are new.

Theorem 3.6. *The adder algorithm expressed by Eq. (1) is correct.*

Proof. Induction on n.
Base case. ($n = 1$). We need to prove that

$$x_0 + y_0 + c_0 = s_0 + 2c_1. \tag{3}$$

By Lemma 3.4, we have:

$$\begin{aligned} x_0 + y_0 + c_0 &= 2(x_0 \wedge y_0) + x_0 \oplus y_0 + c_0 \quad \text{by Lemma 3.4} \\ &= 2(x_0 \wedge y_0) + 2[(x_0 \oplus y_0) \wedge c_0] + x_0 \oplus y_0 \oplus c_0 \quad \text{by Lemma 3.4} \\ &= 2\{(x_0 \wedge y_0) \vee [(x_0 \oplus y_0) \wedge c_0]\} + x_0 \oplus y_0 \oplus c_0 \quad \text{by Lemma 3.5.} \end{aligned}$$

By Exhaustive testing x_0, y_0, c_0, we have:

$$(x_0 \wedge y_0) \vee [(x_0 \oplus y_0) \wedge c_0] = (x_0 \wedge y_0) \vee (x_0 \wedge c_0) \vee (y_0 \wedge c_0).$$

By Eq. (1), we know

$$\begin{aligned} s_0 &= x_0 \oplus y_0 \oplus c_0 \\ c_1 &= (x_0 \wedge y_0) \vee (x_0 \wedge c_0) \vee (y_0 \wedge c_0). \end{aligned}$$

Thus

$$x_0 + y_0 + c_0 = s_0 + 2c_1.$$

Induction case. Assume that Eq. (2) is true for $n = k$:

$$\sum_{i=0}^{k-1} x_i 2^i + \sum_{i=0}^{k-1} y_i 2^i + c_0 = \sum_{i=0}^{k-1} s_i 2^i + c_k 2^k. \tag{4}$$

We need to show that the assertion is valid for $n = k + 1$:

$$\sum_{i=0}^{k} x_i 2^i + \sum_{i=0}^{k} y_i 2^i + c_0 = \sum_{i=0}^{k} s_i 2^i + c_{k+1} 2^{k+1}. \tag{5}$$

Due to the induction hypothesis equation (4), the equality equation (5) is equivalent to

$$x_k 2^k + y_k 2^k = s_k 2^k + c_{k+1} 2^{k+1} - c_k 2^k. \tag{6}$$

Divide 2^k at two sides of the equality, we get

$$x_k + y_k + c_k = s_k + 2c_{k+1}. \tag{7}$$

By Eq. (1) and Lemma 3.4, Lemma 3.5, we have:

$$\begin{aligned} x_k + y_k + c_k &= x_k \oplus y_k \oplus c_k + 2[(x_k \wedge y_k) \vee (x_k \wedge c_k) \vee (x_k \wedge c_k)] \\ &= s_k + 2c_{k+1}. \end{aligned} \tag{8}$$

So, the induction case is proved. □

4. Sequential prefix adders

The critical path of binary adder is the carry computation. It is clear that the long carry propagation chains must be dealt with in order to speed up the addition. The commonly used scheme for accelerating carry propagation is the prefix scheme. The main idea behind prefix addition is rewrite the computation of calculating carries in Eq. (1) as first-order recurrences. This makes it possible to generate all incoming carries in parallel. To define the first-order recurrences form of carry computation, some standard notions such as *propagation carry* and *generated carry* are introduced as in [14]. For two binary numbers $x = (x_{n-1}x_{n-2}\cdots x_0)$ and $y = (y_{n-1}y_{n-2}\cdots y_0)$, the *propagation carry* and *generated carry* at the bit location i are defined as $P_i = x_i \oplus y_i$ and $G_i = x_i \wedge y_i$ separately.

Using *propagation carry*and *generated carry*, the carry and sum computations in Eq. (1) can be rewritten as:

$$c_{i+1} = G_i \vee (P_i \wedge c_i), \qquad s_i = P_i \oplus c_i. \tag{9}$$

If the carries in the Eq. (9) are not unfolded, Eq. (9) corresponds to the Ripple Carry Adder, in which the calculation at bit position i depends on the carry output c_i from previous position $i-1$. Thanks to the first-order recurrence form, by substitution, this type of expression allows us to calculate all carries in parallel from the original digits $x_{n-1}x_{n-2}\cdots x_0$, $y_{n-1}y_{n-2}\cdots y_0$ and the forced carry c_0, which corresponds to the scheme of Carry Lookahead Adder. In this case, the expressions of all c_i in Eq. (9) are unfolded. Since unfolding does not change behavior, the correctness of Eq. (9) implies the correctness of sequential prefix adders schemes including Ripple Carry Adder and Carry Lookahead Adder.

Theorem 4.1. *The adder algorithm expressed by Eq. (9) is correct.*

Proof. We just need to prove the Eq. (9) is equivalent to the Eq. (1). Since $P_i = x_i \oplus y_i$, it is easy to verify that:

$$P_i \oplus c_i = x_i \oplus y_i \oplus c_i.$$

Since $G_i = x_i \wedge y_i$, we get

$$c_{i+1} = G_i \vee (P_i \wedge c_i) = (x_i \wedge y_i) \vee [(x_i \oplus y_i) \wedge c_i].$$

By exhaustive testing binary bits x_i, y_i, c_i, we have:

$$(x_i \wedge y_i) \vee [(x_i \oplus y_i) \wedge c_i] = (x_i \wedge y_i) \vee (x_i \wedge c_i) \vee (y_i \wedge c_i).$$

So, the Eq. (9) is equivalent to the Eq. (1). By Theorem 3.6, the correctness of Eq. (9) is proved. □

For a N-bits Carry Lookahead Adder, if the value of n is large, then, an extremely large number of gates is needed and gates with a very large fan-in are required. One approach is to divide the n stages into groups and have a separate Carry Lookahead Adder in each group. The groups can then be interconnected by other scheme such as Ripple Carry adder. This method reduces the span of the Carry Lookahead Adder at the expense of speed. Other better approaches are parallel prefix adders. To derive the equations for prefix adders in a more general way, we introduce the definition of fundamental carry operator. Our definition of fundamental carry operator is inspired by the first-order recurrences form of carry computation discussed above, but it is at a deeper level. It will allow us to consider various implementations of prefix adders rather than being restricted to a predetermined blocking factor.

Definition 4.2 (*Fundamental Carry Operator*)**.** Let B be a nonempty set of binary bit pairs. The *fundamental carry operator* $\circ$: $B \times B \to B$ is a binary operation such that, for any binary bits a_1, a_2, b_1, b_2, the following equation holds:

$$(a_1, b_1) \circ (a_2, b_2) = (a_1 \vee (b_1 \wedge a_2), b_1 \wedge b_2). \tag{10}$$

The set of bits pairs B and the *fundamental carry operator* over it constitute a nice algebraic structure. We study its algebraic properties in the following.

Lemma 4.3. $(B, \circ)$ *is a semi-group.*

Proof. By the Definition 3.1, we prove that $(B, \circ)$ satisfies its two conditions.

1. For any $(a_1, b_1), (a_2, b_2) \in B$, where a_1, b_1, a_2, b_2 are any binary bits, we have:

$$(a_1, b_1) \circ (a_2, b_2) = (a_1 \vee (b_1 \wedge a_2), b_1 \wedge b_2).$$

It is easy to verify that $a_1 \vee (b_1 \wedge a_2)$ and $b_1 \wedge b_2$ are also binary bits, so, we get $(a_1 \vee (b_1 \wedge a_2), b_1 \wedge b_2) \in B$, which means B is closed under $\circ$.

2. We need to prove that $\circ$ enjoys associativity. For any $(a_1, b_1), (a_2, b_2), (a_3, b_3) \in B$, we have

$$\begin{aligned}
[(a_3, b_3) \circ (a_2, b_2)] \circ (a_1, b_1) &= (a_3 \vee (b_3, a_2), b_3 \wedge b_2) \circ (a_1, b_1) \\
&= (a_3 \vee (b_3 \wedge a_2) \vee (b_3 \wedge b_2 \wedge a_1), b_3 \wedge b_2 \wedge b_1) \\
&= (a_3 \vee (b_3 \wedge (a_2 \vee (b_2 \wedge a_1))), b_3 \wedge b_2 \wedge b_1) \\
&= (a_3, b_3) \circ (a_2 \vee (b_2 \wedge a_1), b_2 \wedge b_1) \\
&= (a_3, b_3) \circ [(a_2, b_2) \circ (a_1, b_1)].
\end{aligned}$$

So, $(B, \circ)$ is a semi-group. □

Lemma 4.4. $(B, \circ)$ *is a monoid.*

Proof. It is known that $(B, \circ)$ is a semi-group. We need to prove that it has an identity element.

Because $(0, 1) \in B$, for any $(a, b) \in B$, we have

$$(a, b) \circ (0, 1) = (a \vee (b \wedge 0), b \wedge 1) = (a, b)$$
$$(0, 1) \circ (a, b) = (0 \vee (1 \wedge a), 1 \wedge b) = (a, b).$$

So, $(0, 1)$ is an identity element. $(B, \circ)$ is a monoid. □

Lemma 4.5. *Every element of* $(B, \circ)$ *is an idempotent.*

Proof. It is known that $(B, \circ)$ is a semi-group. So, for any element $(a, b) \in B$, we just need to prove that $(a, b) \circ (a, b) = (a, b)$. Because a, b are binary bits, we have

$$(a, b) \circ (a, b) = (a \vee (b \wedge a), b \wedge b) = (a, b).$$

So, (a, b) is an idempotent. Without loss of generality, every element of $(B, \circ)$ is an idempotent. □

The *fundamental carry operator* which composes two carry propagation functions (propagation carry and generated carry) is discovered by Brent and Kung [5]. In this paper, the definition of *fundamental carry operator* is given in terms of Boolean operations on bit pairs. We developed some general algebraic properties of our new *fundamental carry operator*. Since carry propagation functions such as propagation carry and generated carry can be represented as bit pairs, our fundamental carry operator can be used to compose them and the algebraic properties are also satisfied. In fact, any functions whose domain an range are both binary bit can be composed by our new *fundamental carry operator*. This is also a good reason for using $\circ$, the standard symbol for function composition, to represent the fundamental symbol for function composition, to represent the *fundamental carry operator*. We believe this insight is central to the operation of prefix adders.

Let *fst* be the function which returns the first element of a pair. Using the definition of $\circ$, the calculation of carries in Eq. (9) can be rewritten as

$$c_{i+1} = fst((G_i, P_i) \circ (c_i, 1)).$$

An important special case is that, for Carry Lookahead Adder, any c_{i+1} which is computed using initial arguments, corresponding to the unfolded form of Eq. (9), can be represented as follows

$$c_{i+1} = fst((G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots (G_0, P_0) \circ (c_0, 1)). \tag{11}$$

Real designs of adder circuits, specially the parallel prefix adders, are often started from the above formula. Because propagation carry and generated carry are functions on binary bits, the algebraic structures we introduced above (semi-group and monoid) and the corresponding algebraic properties such as associativity and idempotent are going to be used to allow the linear carry propagation to be reorganized into group-carry propagation. The full algebraic structure isn't actually needed for the sequential prefix adders. However, it is essential for a general framework that allows a family of parallel prefix adders to be defined. Furthermore, since we have proved the correctness of Carry Lookahead Adder, for more complex parallel prefix adders, if their formulas of calculating carries can be proved to be equivalent to the Eq. (11), then the correctness of their algorithms can be proved effectively.

5. Parallel prefix adders

For sophisticated parallel prefix adders, groups of bits are computed together. To characterize such adders, the notions of *group-propagated carry* and *group-generated carry* have been introduced in the literature, for example [14].

Definition 5.1 (*Group-generated Carry*)**.** Let G_i is the generated carry at bit location i, the group-generated carry $G_{i:j}$ is defined as

$$G_{i:j} = \begin{cases} G_i & i = j \\ G_i \vee (P_i \wedge G_{i-1:j}) & i > j. \end{cases} \tag{12}$$

Definition 5.2 (*Group-propagated Carry*)**.** Let P_i is the propagated carry at bit location i, the group-propagated carry $P_{i:j}$ is defined as

$$P_{i:j} = \begin{cases} P_i & i = j \\ P_i \wedge P_{i-1:j} & i > j. \end{cases} \tag{13}$$

Form the definition of group-generated carry, it is easy to find that the group-generated carry has the first-order recurrences form similar to Eq. (9). Therefore, it is possible to use group-generated carry to describe computation of carries and use fundamental carry operator to represent it. In the following, we give some properties about group-generated carry and group-propagated carry.

Lemma 5.3. $G_{i:j} \wedge P_{i:j} = 0$.

Proof. We need only to consider the case where $i \geq j$. Let $k = i - j$, the proof proceeds by induction on k.

Base case. $k = 0$. $G_{i:j} = G_{i:i} = x_i \wedge y_i$, $P_{i:j} = P_i = x_i \oplus y_i$. So, we have $G_{i:j} \wedge P_{i:j} = G_i \wedge P_i = 0$.

Induction case. Assume that $k = m$, $m \geq 0$, we have $G_{j+m:j} \wedge P_{j+m:j} = 0$, we need to prove $G_{j+m+1:j} \wedge P_{j+m+1:j} = 0$. By Definitions 5.1 and 5.2, we get $G_{j+m+1:j} = G_{j+m+1} \vee (P_{j+m+1} \wedge G_{j+m:j})$, $P_{j+m+1:j} = P_{j+m+1} \wedge P_{j+m:j}$.

Proof by contradiction. Assume $G_{j+m+1:j} \wedge P_{j+m+1} = 1$, then we have $P_{j+m+1} = 1$, $P_{j+m:j} = 1$. By induction assumption $G_{j+m:j} \wedge P_{j+m:j} = 0$, we have $G_{j+m:j} = 0$. Because $G_{j+m+1:j} = G_{j+m+1} \vee (P_{j+m+1} \wedge G_{j+m:j}) = 1$, we have $G_{j+m+1} = 1$. So, we have $P_{j+m+1} = 1$, $G_{j+m+1} = 1$. $P_{j+m+1} \wedge G_{j+m+1} = 1$. By Definition, we know $G_{j+m+1} = x_{j+m+1} \wedge y_{j+m+1}$, $P_{j+m+1} = x_{j+m+1} \oplus y_{j+m+1}$, we have $P_{j+m+1} \wedge G_{j+m+1} = 0$, Contradiction!

So, we get $G_{j+m+1:j} \wedge P_{j+m+1:j} = 0$. The induction case is proved. □

Lemma 5.4. *With fundamental carry operator, group-generated carry and group-propagated carry can be composed as follows:*

$$(G_{i:j}, P_{i:j}) = \begin{cases} (G_i, P_i) & i = j \\ (G_i, P_i) \circ (G_{i-1:j}, P_{i-1:j}) & i > j. \end{cases} \tag{14}$$

Proof. By Definitions 4.2, 5.1 and 5.2, it can be proved immediately. □

The Eq. (14) is a recursive equation. So, it has a flatten representation.

Lemma 5.5 (*group Carry Flatten [11]*)**.**

$$(G_{i:j}, P_{i:j}) = (G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots \circ (G_j, P_j). \tag{15}$$

Since group-generated carry and group-propagated carry are also functions on binary bit, therefore, the fundamental carry operator on pairs of them also enjoys associativity and idempotent. By associativity, the Eq. (14) can be further generalized to

$$(G_{i:j}, P_{i:j}) = (G_{i:m}, P_{i:m}) \circ (G_{m-1:j}, P_{m-1:j}), \quad \text{where } i \geq m \geq j + 1. \tag{16}$$

By idempotent, the most general form of Eq. (16) is

$$(G_{i:j}, P_{i:j}) = (G_{i:m}, P_{i:m}) \circ (G_{v:j}, P_{v:j}), \quad \text{where } i \geq v \geq m - 1 \geq j \geq 0. \tag{17}$$

Due to idempotent, the computation of the left side of the Eq. (17) can be divided into overlapping subgroups which are independent and can be computed in parallel. This gives the possibility to design different implementations of parallel prefix adders.

We have studied some properties about group-generated carry and group-propagated carry above. In the following, we study how to use them to formalize the the algorithms of parallel prefix adders and to prove the correctness. We firstly describe the general algorithm of parallel prefix adders and prove its correctness. It studies the relationship between group-generated carry, group-propagated carry functions and the carries. It is the basis for all parallel prefix adders.

Definition 5.6 (*General Parallel Prefix Adder*)**.** The general parallel prefix adder's algorithms can be presented as follows:

$$c_i = G_{i-1:j} \vee (P_{i-1:j} \wedge c_j), \qquad s_i = P_i \oplus c_i, \quad 0 \leq j \leq i - 1 \leq n - 1. \tag{18}$$

Lemma 5.7. *The general parallel prefix adder's algorithm is correct.*

Proof. To prove the correctness of the Eq. (18), we just need to prove it is equivalent to Eq. (9). Since the main difference of them is the computation of carries, let $c'_i = G_{i-1:j} \vee (P_{i-1:j} \wedge c'_j)$ and $c_i = G_{i-1} \vee (P_{i-1} \wedge c_{i-1})$, we should to prove that

$$c'_i = c_i.$$

Induction on j.

Base case. $j = 0$. Since both c'_0 and c_0 denote the incoming carry into the adder, therefore, we have $c'_0 = c_0$. Then for any i, where $j \leq i - 1 \leq n - 1$, we prove that $c'_i = c_i$ by induction on $k = i - 1 - j$. For the base case $k = 0$, we have $i = j + 1$, $c'_i = c'_{j+1} = G_{j:j} \vee (P_{j:j} \wedge c'_j) = G_j \vee (P_j \wedge c'_0)$. We also know that $c_i = c_{j+1} = G_j \vee (P_j \wedge c_j) = G_j \vee (P_j \wedge c_0)$. Therefore, $c'_i = c_i$. Now, consider the induction case. Assume that $c'_i = c_i$ is true for $k = m$, $m \geq 1$, we want to show that it is true for $k = m+1$. When $k = m+1$, we have $c'_i = c'_{m+1+j+1} = G_{m+j+1:j} \vee (P_{m+j+1:j} \wedge c'_j)$, $c_i = c_{m+1+j+1} = G_{m+j+1} \vee (P_{m+j+1} \wedge c_{m+j+1})$. By the definition of group-generated carry, group-propagated carry and induction hypothesis, we get

$$\begin{aligned} c'_{m+1+j+1} &= G_{m+j+1:j} \vee (P_{m+j+1:j} \wedge c'_j) \\ &= G_{m+j+1} \vee (P_{m+j+1} \wedge G_{m+j:j}) \vee (P_{m+j+1} \wedge P_{m+j:j} \wedge c'_j) \\ &= G_{m+j+1} \vee (P_{m+j+1} \wedge (G_{m+j:j} \vee (P_{m+j:j} \wedge c'_j))) \\ &= G_{m+j+1} \vee (P_{m+j+1} \wedge c_{m+j+1}) \\ &= c_{m+1+j+1}. \end{aligned}$$

So, $c'_i = c_i$ is true for $j = 0$.

Induction case. Assume $c'_i = c_i$ is true for $j < p$, $p \geq 0$, we need to prove that it is also true for $j = p$.

When $j = p$, induction on $k = i - 1 - j$. For the base case $k = 0$, we have $i = j + 1$, $c'_i = c'_{j+1} = G_{j:j} \vee (P_{j:j} \wedge c'_j) = G_j \vee (P_j \wedge c'_j) = G_p \vee (P_p \wedge c'_p)$.
We also know that $c_i = c_{j+1} = G_j \vee (P_j \wedge c_j) = G_j \vee (P_j \wedge c_j) = G_p \vee (P_p \wedge c_p)$. By Eq. (18), we know that c'_p can only be computed by equation $c'_p = G_{p:q} \vee (P_{p:q} \wedge c'_q)$, where $0 \leq q < p$. By induction assumption, we know that $c'_p = c_p$. So, the base case is proved. Now, consider the induction case. Assume that $c'_i = c_i$ is true for $k = m$, $m \geq 1$. We want to show that it is true for $k = m + 1$. The prove process is similar to the base case ($j = 0$). By the definition of group carry and induction hypothesis, we get

$$\begin{aligned} c'_{m+1+j+1} &= G_{m+j+1:j} \vee (P_{m+j+1:j} \wedge c'_p) \\ &= G_{m+j+1} \vee (P_{m+j+1} \wedge G_{m+j:j}) \vee (P_{m+j+1} \wedge P_{m+j:j} \wedge c'_p) \\ &= G_{m+j+1} \vee (P_{m+j+1} \wedge (G_{m+j:j} \vee (P_{m+j:j} \wedge c'_p))) \\ &= G_{m+j+1} \vee (P_{m+j+1} \wedge c_{m+j+1}) \\ &= c_{m+1+j+1}. \end{aligned}$$

So, the induction case is proved. □

Fundamental carry operator can be used to rewrite the computation of carry in Eq. (18) as follows:

$$c_{i+1} = fst((G_{i:j}, P_{i:j}) \circ (c_j, 1)). \tag{19}$$

According to Eq. (15), we can give the flatten representation of Eq. (19) as follows:

$$c_{i+1} = fst((G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots \circ (G_j, P_j) \circ (c_j, 1)). \tag{20}$$

Since the incoming carry c_0 is always known at the beginning, any carry can be computed using the initial arguments in parallel. Therefore, the Eq. (20) can be rewritten as:

$$c_{i+1} = fst((G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots \circ (G_0, P_0) \circ (c_0, 1)).$$

The above formula is same to Eq. (11). Therefore, we say it is the base for design and correctness proof of prefix adders.

At last, we give a correctness proof of a special parallel prefix adder named Kogge–Stone tree to show the effectiveness of our approach. Using the notions of fundamental carry operator, group-generated carry and group-propagated carry, we formalized the algorithm of Kogge–Stone tree in a new way. Then, we developed a new proof for the correctness of Kogge–Stone tree. The algebraic properties we developed above are helpful for our proof. Compared to the related works, our concise proof is shorter and easy to read.

Definition 5.8. Let c_i denote the carry from the $i - 1$ bit position. To compute c_i, Kogge–Stone tree needs $\log_2(i + 1)$ steps to calculate the corresponding group-generated carry and group-propagated carry. Let (G_i^k, P_i^k) represent respectively, the group-generated carry and group-propagated carry after the kth step of the following algorithm.

Initialization Step ($k = 0$):

$$G_i^0 = G_i, \qquad P_i^0 = P_i. \tag{21}$$

Recursion steps: For $k = 1, 2, \ldots, \lceil \log_2(i + 1) \rceil$ do each of the following assignment statements:

$$\begin{aligned} &(G_i^k, P_i^k) = (G_i^{k-1}, P_i^{k-1}) \circ (G_{i-2^{k-1}}^{k-1}, P_{i-2^{k-1}}^{k-1}) \\ &(G_i^m, P_i^m) = (G_i^{\lceil \log_2(i+1) \rceil}, P_i^{\lceil \log_2(i+1) \rceil}) \quad \lceil \log_2(i + 1) \rceil < m. \end{aligned} \tag{22}$$

Lemma 5.9. *For any i and $1 \leq k$, if $k \leq \lceil \log_2(i + 1) \rceil$, then*

$$(G_i^k, P_i^k) = (G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots \circ (G_{i-(2^k-1)}, P_{i-(2^k-1)}). \tag{23}$$

Proof. Because $\circ$ is associative, we give our proof by induction on k.
Base case. $k = 1$, for any $i \geq 1$, since $\log_2(i + 1) \geq \log_2 2 = 1$, by Eq. (22), we have:

$$\begin{aligned} (G_i^1, P_i^1) &= (G_i^0, P_i^0) \circ (G_{i-2^0}^0, P_{i-2^0}^0) \\ &= (G_i^0, P_i^0) \circ (G_{i-(2^k-1)}^0, P_{i-(2^k-1)}^0). \end{aligned}$$

By Eq. (21), we get:

$$(G_i^1, P_i^1) = (G_i, P_i) \circ (G_{i-1}, P_{i-1}).$$

Induction case. Assume that (23) is true for $k < p$, we need to show that the assertion is valid for $k = p$, where $p \leq \lceil \log_2(i + 1) \rceil$.

By Eq. (22), we have:

$$(G_i^p, P_i^p) = (G_i^{p-1}, P_i^{p-1}) \circ (G_{i-2^{p-1}}^{p-1}, P_{i-2^{p-1}}^{p-1}).$$

By induction hypothesis, we get:

$$(G_i^{p-1}, P_i^{p-1}) = (G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots \circ (G_{i-(2^{p-1}-1)}, P_{i-(2^{p-1}-1)}). \tag{24}$$

(1) If $p-1 \leq \lceil \log_2(i-2^{p-1}+1) \rceil$, by induction hypothesis, we have

$$(G_{i-2^{p-1}}^{p-1}, P_{i-2^{p-1}}^{p-1}) = (G_{i-2^{p-1}}, P_{i-2^{p-1}}) \circ \cdots \circ (G_{i-2^{p-1}-(2^{p-1}-1)}, P_{i-2^{p-1}-(2^{p-1}-1)}).$$

So, we get

$$(G_i^p, P_i^p) = (G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots \circ (G_{i-(2^p-1)}, P_{i-(2^p-1)}).$$

(2) If $p-1 > \lceil \log_2(i-2^{p-1}+1) \rceil$, by Eq. (22), we have:

$$(G_{i-2^{p-1}}^{p-1}, P_{i-2^{p-1}}^{p-1}) = (G_{i-2^{p-1}}^{\lceil \log_2(i-2^{p-1}+1) \rceil}, P_{i-2^{p-1}}^{\lceil \log_2(i-2^{p-1}+1) \rceil}).$$

By induction hypothesis, we have

$$(G_{i-2^{p-1}}^{\lceil \log_2(i-2^{p-1}+1) \rceil}, P_{i-2^{p-1}}^{\lceil \log_2(i-2^{p-1}+1) \rceil}) = (G_{i-2^{p-1}}, P_{i-2^{p-1}}) \circ \cdots \circ (G_{i-2^{p-1}-(2^{p-1}-1)}, P_{i-2^{p-1}-(2^{p-1}-1)}).$$

So, we get

$$(G_i^p, P_i^p) = (G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots \circ (G_{i-(2^p-1)}, P_{i-(2^p-1)}).$$

The induction case is proved. □

Theorem 5.10. *The algorithm of Kogge–Stone tree described in Definition 5.8 is correct.*

Proof. We just need to prove

$$(G_i^{\lceil \log_2(i+1) \rceil}, P_i^{\lceil \log_2(i+1) \rceil}) = (G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots \circ (G_0, P_0). \tag{25}$$

Because $\circ$ is associative, we give our proof by induction on i.
Base case. $i = 1$, $\lceil \log_2(i+1) \rceil = \lceil \log_2 2 \rceil = 1$. We get

$$\begin{aligned}(G_i^{\lceil \log_2(i+1) \rceil}, P_i^{\lceil \log_2(i+1) \rceil}) &= (G_1^{\lceil \log_2(2) \rceil}, P_1^{\lceil \log_2(2) \rceil}) \\ &= (G_1^1, P_1^1) \\ &= (G_1^0, P_1^0) \circ (G_0^0, P_0^0) \quad \text{by Eq. (22)} \\ &= (G_1, P_1) \circ (G_0, P_0) \quad \text{by Eq. (21).}\end{aligned} \tag{26}$$

Induction case.
Assume that Eq. (25) is true for $i < m$, $m > 1$. We need to show that the assertion is valid for $i = m$:

$$(G_m^{\lceil \log_2(m+1) \rceil}, P_m^{\lceil \log_2(m+1) \rceil}) = (G_m, P_m) \circ (G_{m-1}, P_{m-1}) \circ \cdots \circ (G_0, P_0). \tag{27}$$

Let $k = \lceil \log_2(m+1) \rceil$, because $m > 1$, we know that $k > 1$. We have: $(G_m^{\lceil \log_2(m+1) \rceil}, P_m^{\lceil \log_2(m+1) \rceil}) = (G_m^k, P_m^k)$. By Eq. (22), we get:

$$(G_m^k, P_m^k) = (G_m^{k-1}, P_m^{k-1}) \circ (G_{m-2^{k-1}}^{k-1}, P_{m-2^{k-1}}^{k-1}).$$

By Lemma 5.9, we have:

$$(G_m^{k-1}, P_m^{k-1}) = (G_m, P_m) \circ (G_{m-1}, P_{m-1}) \circ \cdots \circ (G_{m-(2^{k-1}-1)}, P_{m-(2^{k-1}-1)}). \tag{28}$$

Since $\log_2(m-2^{k-1}+1) = \log_2(m+1) - (k-1)$, we have

$$\lceil \log_2(m-2^{k-1}+1) \rceil = \lceil \log_2(m+1) \rceil - (k-1) = 1.$$

Since $k \geq 2$, we get $k-1 \geq \lceil \log_2(m-2^{k-1}+1) \rceil$
By Eq. (22), we get

$$(G_{m-2^{k-1}}^{k-1}, P_{m-2^{k-1}}^{k-1}) = (G_{m-2^{k-1}}^{\lceil \log_2(m-2^{k-1}+1) \rceil}, P_{m-2^{k-1}}^{\lceil \log_2(m-2^{k-1}+1) \rceil}).$$

By induction hypothesis, we have

$$(G^{\lceil \log_2(m-2^{k-1}+1)\rceil}_{m-2^{k-1}}, P^{\lceil \log_2(m-2^{k-1}+1)\rceil}_{m-2^{k-1}}) = (G_{m-2^{k-1}}, P_{m-2^{k-1}}) \circ (G_{m-2^{k-1}-1}, P_{m-2^{k-1}-1}) \circ \cdots \circ (G_0, P_0). \tag{29}$$

By Eq. (28) and Eq. (29), we get

$$\begin{aligned}(G^k_m, P^k_m) &= (G^{k-1}_m, P^{k-1}_m) \circ (G^{k-1}_{m-2^{k-1}}, P^{k-1}_{m-2^{k-1}}) \\ &= (G_m, P_m) \circ \cdots \circ (G_{m-(2^{k-1}-1)}, P_{m-(2^{k-1}-1)}) \circ (G_{m-2^{k-1}}, P_{m-2^{k-1}}) \circ (G_{m-2^{k-1}-1}, P_{m-2^{k-1}-1}) \cdots \circ (G_0, P_0) \\ &= (G_m, P_m) \circ (G_{m-1}, P_{m-1}) \circ \cdots \circ (G_0, P_0). \end{aligned} \tag{30}$$

The induction case is proved. □

Group carry functions can be represented as bit pairs, which compose semi-group with fundamental carry operator. Our proof is benefit from properties of these algebraic structures. For Kogge–Stone tree, the most important property is associativity. Other parallel prefix trees will need other properties such as idempotency. The fundamental carry operator, standard carry functions and the algebraic structures developed here can be used as a general proof framework for parallel prefix adders. In this paper, we focus on the correctness proof of prefix adders. Lakshmivarahan and Dhall [12] built a semi-group for carry computation of prefix adders in a different way. They analyzed other formal properties of prefix adders with the help of algebraic properties of semi-group. For example, group-free semi-groups play a vital role in the design of unbounded fan-in, constant depth and polynomial size prefix circuits [5]. Likewise, the size of circuit depends on whether or not the underlying semi-group is cycle-free [5]. For our further work, we will focus on analyzing other formal properties of prefix adders based on our mathematical machinery.

6. Conclusion

In this paper, we present a formal method to represent prefix adders in terms of first-order recursive equations used in standard computer arithmetic community. We integrate some well known formal properties of prefix adders and introduce a new definition fundamental carry operator. We also prove that the new fundamental carry operator and the set of binary bits compose semi-group, which is helpful to characterize the adder algorithms. With these algebraic structures, both sequential and parallel computations are formalized and proved. This correctness study provides the underpinnings and insights for devising more efficient adders.

References

[1] M.J. Flynn, S.F. Oberman, Modern research in computer arithmetic, Class notes, Stanford Univ. Stanford, CA, Autumn quarter, 1998–1999.
[2] T. Lynch, J. Earl, E. Swartzlander, A spanning tree carry lookahead adder, IEEE Trans. Comput. 41 (8) (1992) 931–939.
[3] R. Ladner, M. Fischer, Parallel prefix computation, J. ACM 27 (4) (1980) 831–838.
[4] P.M. Kogge, H.S. Stone, A parallel algorithm for the efficient solution of a general class of recurrence equations, IEEE Trans. Comput. C-22 (8) (1973) 786–793.
[5] R.P. Brent, H.T. Kung, A regular layout for parallel adders, IEEE Trans. Comput. C-31 (2) (1982) 260–264.
[6] S. Knowles, A family of adders, in: 15th IEEE Symposium on Computer Arithmetic, 2001, pp. 277–281.
[7] J. O'Donnell, G. Rnger, Derivation of a carry lookahead addition circuit, J. Funct. Programming 14 (6) (2004) 127–158.
[8] D. Kapur, M. Subramaniam, Mechanical verification of adder circuits using rewrite rulelaboratory, Form. Methods Syst. Des. 13 (2) (1998) 127–158.
[9] M. Sheeran, Hardware design and functional programming: A perfect match, J. UCS 11 (7) (2005) 1135–1158.
[10] R. Hinze, An algebra of scans, in: Dexter Kozen (Ed.), Proceedings of the Seventh International Conference on Mathematics of Program Construction, MPC 2004, 2004, pp. 186–210.
[11] Gang Chen, Feng Liu, Proofs of correctness and properties of integer adder circuits, IEEE Trans. Comput. 59 (1) (2010) 134–136.
[12] S. Lakshmivarahan, S.K. Dhall, Parallel Computing Using the Prefix Problem, Oxford University Press, USA, 1994.
[13] J.M. Howie, Fundamentals of Semigroup Theory, Oxford University Press, USA, 1996.
[14] I. Koren, Computer Arithmetic Algorithms, 2nd edition, A.K.Peters, Natick, MA, 2002.