Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 197252, 9 pages
http://dx.doi.org/10.1155/2014/197252

Research Article

Functional Verification of High Performance Adders in Coq

Qian Wang,[1] Xiaoyu Song,[2] Ming Gu,[1] and Jiaguang Sun[1]

[1] *School of Software, Tsinghua University, Beijing 100084, China*
[2] *Department of ECE, Portland State University, Portland, OR 97207, USA*

Correspondence should be addressed to Qian Wang; qw04ster@gmail.com

Received 3 January 2014; Accepted 10 February 2014; Published 10 April 2014

Academic Editor: Krishnaiyan Thulasiraman

Addition arithmetic design plays a crucial role in high performance digital systems. The paper proposes a systematic method to formalize and verify adders in a formal proof assistant Coq. The proposed approach succeeds in formalizing the gate-level implementations and verifying the functional correctness of the most important adders of interest in industry, in a faithful, scalable, and modularized way. The methodology can be extended to other adder architectures as well.

1. Introduction

Demonstrating the functional correctness of an arithmetic implementation is a challenging topic which has lasted for several decades. Testing and simulation, as the traditional methods, have won good reputation and have been employed extensively in industry. When dealing with large scale designs, these methods may find counterexamples but could not assert if a design is correct because the exhaustivity is impractical.

As an alternative, formal methods have been increasingly adopted to validate the arithmetic implementations. A main branch of formal methods is model checking, which is recognised by its automation and succeeds in numerous industrial applications. However, the inherent state explosion problem prevents it from scaling to large scale designs.

Another branch of verification is theorem proving, which is no longer restricted by the scale as model checking, testing, and simulation. The main problem restricting theorem proving to be widespread is that it requires strong logic backgrounds and heavy user interactions. Nevertheless, there appear quite a few successful applications by different theorem provers. By Boyer-Moore, a microprocessor is verified in [1], and an N-bit comparator as well as mean-value circuits are verified in [2]. By HOL, a ripple carry adder and a sequential device are verified in [3], and an ATM switch fabric is verified in [4]. By Coq, a sequential multiplier is verified in [5], and an asynchronous transfer mode switch fabric is verified in [6].

The main effort of this work is to propose a holistic methodology to formalize and verify adders in Coq [7]. Adders are chosen because they are the most fundamental arithmetic units widely employed in various advanced digital systems, such as IBM POWER6, whose correctness depends significantly on the correctness of its addition subcomponents. This methodology provides a uniform way to formalize and verify various implementations of arithmetic addition, and it is applied in this work to formalize and verify primary and high speed adders of interest in industry, including Carry Look-ahead Adder (CLA), Ling Adder (LA), and Parallel Prefix Adder (PPA).

Benefiting from the techniques of Coq, the methodology shares the following decent features.

(i) Scalability: the formalization of an adder is parameterized by a natural number (named *length*) and the correctness proof applies to any length.

(ii) Modularization: various verified adders are encapsulated as instances of an abstract module, which provides a uniform way to be reused in advanced arithmetic units. The formalization and verification of an advanced arithmetic unit can be accumulated from verified units ignoring their detailed implementations.

(iii) Fidelity: the adders are formalized by (recursive) functions, which have clear correspondences to the gate-level implementations of circuits. The addends

and sum of an adder are formalized as vectors, which is a faithful model of arrays and provides meanwhile additional type checking ability to avoid potential misusing of inputs.

The rest of paper is organized as follows. Related works are introduced in Section 2. According to our knowledge, we verify not only most adders appearing in the literature, but also some for the first time by theorem proving. Section 3 explains our methodology in details by the example of ripple carry adder. Preliminaries are also introduced according to our needs. Some definitions and most proofs will not be presented in this paper, but they are available on the author's webpage (http://superwalter.github.io/dev/veriadder.zip). Sections 4 and 5 are devoted to LA and PPA, respectively.

2. Related Work

Compared to their extensive applications, the verification of primary adders by theorem proving is not at the fingertips. In particular, the formalization and verification of the Ling adder cannot be found in any literature. Reference [8] proves the correctness of RCA by formalizing adders with dependent types in Coq. Reference [9] proves the correctness of RCA by the higher-order logic with a reusable library for formalizing circuits. Reference [2] verifies RCA written in VHDL as well as other circuits by the higher-order logic. Reference [10] develops semiformal correctness proof of CLA or PPA. Reference [11] shows a pencil-and-paper proof of the general prefix adders, as well as the proof of related RCA. Furthermore, [12] formalizes and verifies these adders in Coq. By rewriting and induction, [13] provides the verification of PPA using powerlists. An algebra formalization of PPA and its correctness proof are presented in [14]. Besides applying it to formalize and verify most primary adders, our methodology also provides good features, which only appear partially in other literatures, but are never integrated together in any preview work, according to our knowledge.

3. A Holistic Methodology

Various kinds of adders are designed to provide relatively good performances for different circumstances, while they implement the same addition functionality. A holistic methodology is proposed in this work in order to capture all the different adders and provide desired good features.

3.1. A Unified Proof Structure. Basically, the methodology answers four questions:

(i) how to formalize the related data types;

(ii) which method is used to formalize an adder;

(iii) what should be proved;

(iv) how to organize formalizations and verifications for different adders.

These questions are answered by a uniform specification, utilizing the module system of Coq.

```
(1) Definition mbadder (n: nat):=
(2) data (S n)-> data (S n)-> bit->
    hyb (S n).
(3) Definition mbadder_c n (f: mbadder n):=
(4) forall (X Y: data (S n)) c,
(5) |[X]| + |[Y]| + |c| = |(f X Y c)|.
(6) Module Type GenAdder.
(7) Parameter adder: forall n, mbadder n.
(8) Axiom adder_correct: forall (n:nat),
(9) mbadder_correct (@adder n).
(10) End GenAdder.
```

Lines 1 and 2 answer the first two questions. n, in line 1, is a parameter (name *length*) indicating the inherent nature of an adder: how many bits it can process. The input carry-in and output carry-out are formalized by Booleans (`bit`). The input addends and the returned sum are formalized by vectors of Booleans (`data` m), which are dependent types depending on the length m. `hyp` m is another dependent type standing for a tuple of a bit and a m-bit vector, which is used in line 2 for combining the carry-out and the sum. Thus, an adder is formalized as a function, taking two addends and a carry-in as inputs and returning a tuple of carry-out and sum. This function is normally recursively defined as shown later.

Lines 3, 4, and 5 answer the third question. The correctness of an adder is ensured by proving that the natural number denotations of the inputs and outputs are equivalent. In line 5, $|b|$ is the natural number denotation of a bit b. $|[v]|$ and $|(t)|$ are natural number denotations of the vector v and the result tuple t. Big endian is chosen to implement these two functions.

Lines 6–10 answer the last question. A general adder is formalized as an abstract module. The specification is assigned and the correctness is required. A verified adder should be its instance, like a Ripple Carry Adder (RCA).

3.2. An Example Explaining the Methodology. Carry Look-ahead Adder (CLA) improves RCA by computing all the carries in advance in order to reduce the significant delay. This is represented, in the formalization, by extending the general module with abstract functions P, G, and *carries* which are supposed to compute all the propagated carries, generated carries, and carries, respectively, according to the inputs.

```
(1) Module Type LookAheadAdder <: GenAdder.
(2) Parameter P: forall n, data n -> data
    n -> data n.
(3) Parameter G: forall n, data n -> data
    n -> data n.
(4) Parameter carries n: data (S n) -> data
    (S n) -> bit -> hyb (S n).
(5) Parameter adder: forall n, mbadder n.
```

(6) `Parameter adder_correct: forall n, mbadder_correct (@adder n).`

(7) `End LookAheadAdder.`

<: symbol in line 1 stands for the fact that this module should be an instance of the general verified adder. RCA is formalized according to the following equations:

$$c_{i+1} = (x_i \wedge y_i) \vee ((x_i \oplus y_i) \wedge c_i) = g_i \vee (p_i \wedge c_i), \quad (1)$$

$$s_i = x_i \oplus y_i \oplus c_i = p_i \oplus c_i. \quad (2)$$

Carry to each bit c_{i+1} in CLA is computed by iteratively unfolding c_i in (1) until c_0 which is an overall input bit as shown by the following example:

$$\begin{aligned} c_3 &= g_2 \vee (p_2 \wedge c_2) \\ &= g_2 \vee (p_2 \wedge (g_1 \vee (p_1 \wedge c_1))) \\ &= g_2 \vee (p_2 \wedge g_1) \vee (p_2 \wedge p_1 \wedge c_1) \\ &= g_2 \vee (p_2 \wedge g_1) \vee (p_2 \wedge p_1 \wedge (g_0 \vee (p_0 \wedge c_0))) \\ &= g_2 \vee (p_2 \wedge g_1) \vee (p_2 \wedge p_1 \wedge g_0) \vee (p_2 \wedge p_1 \wedge p_0 \wedge c_0). \end{aligned} \quad (3)$$

This process as well as definitions of P and G are formalized as follows:

(1) `Definition P n (X Y: data n):= X ⊕ Y.`

(2) `Definition G n (X Y: data n):= X ∧ Y.`

(3) `Definition carries n (X Y: data (S n)) (cin: bit): hyb (S n).`

(4) `induction n as [|n rec].`

(5) `+ exact (bandor (Y ▷) (X ▷) cin, [cin]).`

(6) `+ set (recs:= rec (X ◁) (Y ◁)).`

(7) `exact (bandor (Y ▷) (X ▷) (recs`$_1$`), (recs`$_1$`)⋈(recs`$_2$`)).`

(8) `Defined.`

⊕ and ∧ in lines 1 and 2 and ∨ used later are extensions of logical Boolean operations ⊕, ∧, and ∨, iterating these operations on the elements at the same position of the two vectors. + symbols in lines 5 and 6 stand for the start of the two branches of the recursion where $n = 0$ or $n = m + 1$. The ▷ operators in line 5 return the leftmost element of a vector. Correspondingly, the ◁ operator in line 6 returns the rightmost n elements of a $(n+1)$-bit vector. $[b]$ is a vector with a single bit b. p_1 and p_2 represent the first and second objects of a tuple, respectively. The ⋈ operator in line 9 joins a bit and a n-bit vector to form a $(n + 1)$-bit vector.

The adder is defined as follows and its correctness is proved by induction on the length and reusing the correctness result of the full adder:

(1) `Definition adder: forall n, mbadder n.`

(2) `intros n X Y cin.`

(3) `set (cc:= carries (P X Y) (G X Y) cin).`

(4) `exact (cc`$_1$`, (P X Y) ⊕ (cc`$_2$`)).`

(5) `Defined.`

(6) `Theorem adder_correct: forall n, mbadder_correct (@adder n).`

(7) `Proof. induction n as [|n rec]. ... Qed.`

3.3. Features Provided by the Methodology. There are several benefits to the use of this methodology for the verification of adders.

3.3.1. Scalability. The formalization and verification of an adder is scalable to any data-width, because the parameterized length can be specified to arbitrary natural number. A 4-bit RCA can be obtained by the following:

(1) `Definition CLA4:= CLA 3.`

(2) `Corollary CLA4_correct: forall (X Y: data 4) c,`

(3) `|[X]| + |[Y]| + |c| = |(RCA4 X Y c)|.`

(4) `Proof. intros; apply CLA_correct. Qed.`

Notice that a 4-bit CLA is `CLA3`, because we require that the addends of the adders have at least one bit. The correctness proof of a CLA with a specified length follows straightforwardly from the proof of CLA with arbitrary length.

3.3.2. Modularization. Some high speed adders divide the input addends into different groups. Each group is calculated by a Carry Selected Adder (CSA) independently, and different groups will be concatenated together in order. Since the computation of CSA depends on the very late steps of input carry-in, such designs would have less propagated time, thus high performance. We formalize an abstract architecture for this kind of design, which illustrates the modularization of our method and may also contribute to verify complex adders in the future.

CSA takes an abstract verified adder as parameter and is also an instance of the general verified adder.

(1) `Module CSA (M: GenAdder) <: GenAdder.`

(2) `Definition adder n: mbadder n.`

(3) `intros X Y c.`

(4) `set (a1:= M.adder X Y true).`

(5) `set (a0:= M.adder X Y false).`

(6) `set (sum:= (dmap (band c) a1`$_2$`) ∨ (dmap (band (¬c)) a0`$_2$`)).`

(7) `set (c':= (a1`$_1$` ∧ c) ∨ (a0`$_1$` ∧ (¬c))).`

(8) `exact (c', sum).`

(9) `Defined.`

```
(10) Theorem adder_c: forall n, badder_
     correct (@adder n).
(11) Proof. ... rewrite M.adder_c. ... Qed.
(12) End CSA.
(13) Module CSA_CLA:= CSA CLA.
```

Lines 2 to 10 define CSA. Two adders compute the sum and the carry-out with respect to carry-in *true* and *false* in lines 4 and 5, respectively. The multiplexer chooses the real sum and carry-out according to the actual carry-in in lines 6 and 7, since when the input carry is required. *dmap* in line 6 applies a function to each element of a vector. The correctness of CSA holds because the addition unites are correct; thus, CSA is an instance of the general adder. The parameterized module can be instantiated by any verified adders. Line 13 defines a CSA whose addition unites are specified to CLA.

```
(1) Module Type GroupAdder (M: GenAdder)
    <: GenAdder.
(2) Parameter part: list nat.
(3) Fixpoint adder_rec (n lens len: nat):
    (mbadder lens).
(4) destruct n.
(5) + exact (@M.adder lens).
(6) + specialize adder_rec with (1:=n)
(7) (lens:= pred (cur_index_abr n len))
    (2:=len).
(8) ....
(9) exact (cast_comb (combination
(10) (@M.adder (lens - (cur_index_abr n
     len)))
(11) (adder_rec)) (aux _ _ Hc3 Hc2)).
(12) Defined.
(13) Definition adder n:= adder_rec (sect n)
     n n.
(14) Lemma adder_correct: forall n, mbadder_
     correct (adder n).
(15) Proof.   ...   Qed.
(16) End GroupAdder.
```

The formalization and verification of this adder are quite complex due to the problem with the dependent types as described in [15, 16]; therefore, the unimportant details are omitted. The *part* in line 2 is a partition of the addends. This partition should be valid, which means the elements preserve strict order and do not exceed the total data-width. Lines 3 to 12 define the adder recursively by combining an adder with another which is combination of the remaining groups of adders obtained by recursion. *combination* in line 9 execute the combining operation. *cast_comb* in line 9 converts an adder with length m to an adder with length n taking the proof of $m = n$ as an argument. The initial values of this recursive function are specified in line 13. The correctness can be proved by the induction on the length of the partition and using the correctness result of combining correct adders.

The parameterized module can be instantiated by any verified adder. If it is instantiated by CSA, it is a verification of many popular high speed adders.

3.3.3. Fidelity. There are normally two ways to formalize the addends and sum of an adder in Coq, either by dependent type *vector* as in [6, 8] and this work or nondependent type *list* as in [12]. Both [6, 8] have explanations why dependent type is more proper for the verification of adders. Generally speaking, nondependent list is more proper for formalizing linked list, whose length can be obtained by computation, while dependent vector is more proper for formalizing array, whose length is inherent natural. The functionality of adders is formalized by interactively defined (recursive) functions which have clear correspondences to gate-level description of circuits.

4. Ling Adder

The Ling Adder (LA) was proposed by [17]. Instead of computing in advance all the *carries* as CLA, LA computes all the pseudo carries, the propagation of which have less fan-ins and fan-outs. With the proper grouping of the input addends, LA needs lesser levels of gates and consequently has better performance.

Similar to the propagated and generated carries, LA has new complementing signal k_i and previous stage propagate T_i, which are defined in (4) and (5) respectively as follows:

$$k_i = a_i \wedge b_i, \tag{4}$$

$$T_i = a_i \vee b_i. \tag{5}$$

The pseudo carries are defined recursively. According to our knowledge, [17] and other materials about LA define the pseudo carries without considering the case $i = 0$ as this paper does in (6b).

Consider

$$H_i = k_i \vee (H_{i-1} \wedge T_{i-1}) \quad i > 0, \tag{6a}$$

$$H_i = k_i \vee c_{in} \quad i = 0. \tag{6b}$$

Without this case, the default values of H_{-1} and T_{-1} are both *false*, and it is equivalent to our definition assuming that c_{in} is always *false*. More intuitively, that algorithm does not consider the carry-in to the least significant bit,, which restricts it to some special applications, such as the addition of two registers. We generalize it to provide general functionality of an adder. Sum is defined similarly to consider the carry-in to the least significant bit as follows:

$$s_i = (H_i \oplus T_i) \vee (k_i \wedge H_{i-1} \wedge T_{i-1}) \quad i > 0, \tag{7a}$$

$$s_i = (H_i \oplus T_i) \vee (k_i \wedge c_{in}) \quad i = 0. \tag{7b}$$

The abstract module of Ling extends the general one by adding signatures of k, T, and H.

```
(1) Module Type LingAdder <: GenAdder.
(2) Parameter K: forall n, data n -> data
    n -> data n.
(3) Parameter T: forall n, data n -> data
    n -> data n.
(4) Parameter H: forall n, data (S n)-> data
    (S n)-> bit-> data (S n).
(5) Parameter adder: forall n, mbadder n.
(6) Parameter adder_correct: forall n,
    mbadder_correct (@adder n).
(7) End LingAdder.
```

To compute the ith pseudo carry of H, the ith bit of K and the $(i-1)$th bit of T are needed. Therefore, the two parameters of H stand for vectors K and a left shift of T. The formalization of H assuming the correctness of the parameters is as follows:

```
(1) Definition H n (X Y: data (S n)):
    data (S n).
(2) induction n as [|n rec].
(3) + exact ([(X ▷) ∨ (Y ▷)]).
(4) + set (recs:= rec (X ◁) (Y ◁)).
(5) exact ((X ▷) ∨ ((Y ▷) ∧ (recs ▷))
    ⋈ recs).
(6) Defined.
```

H is defined recursively. Line 3 deals with the case $i = 0$. Lines 4 and 5 deal with the recursive case. *recs* is the last n bits of H by recursion, and *recs* ▷ stands for H_{n-1}.

LA is defined according to (7a) and (7b) using the definition of H.

```
(1) Definition adder n
(2) (X Y: data (S n)) (cin: bit): hyb (S n).
(3) set (KXY:= K X Y).
(4) set (TXY:= T X Y).
(5) set (Tshft:= shiftin cin TXY).
(6) set (Hc:= H KXY (Tshft ◁)).
(7) set (Hcshft:= shiftin true pc).
(8) set (sum:= (TXY ⊕ Hc) ∨ (KXY ∧
    (Hcshft ◁) ∧ (Tshft ◁))).
(9) exact ((TXY ▷) ∧ (Hc ▷), sum).
(10) Defined.
```

Since the ith bit of sum depends on the $(i-1)$th bit of H and T, they are shifted in lines 5 and 7. The reason why *cin* is shifted into T is explained above; *true* is shifted into H to ensure $T_{-1} \wedge H_{-1} = cin$ where H_{-1} and T_{-1} are the bits to be shifted in, respectively, and $T_{-1} = cin$. The carry-out of LA is (TXY ▷) ∧ (Hc ▷) which is equivalent to c_{out} as shown in

$$c_i = H_{i-1} \wedge T_{i-1}, \quad i \geq 0. \tag{8}$$

The formalization of (8) is complicated, but the proof is trivial by induction and case analysis. The correctness of LA follows by proving a lemma stating that the outputs of CLA and LA are the same with regard to arbitrary same inputs. This lemma is proved by induction with the result of (8).

```
(1) Lemma LA_CLA_eq: forall n (X Y: data
    (S n)) c_in,
(2) LAdder.adder X Y c_in = CLAdder.adder
    X Y c_in.
(3) Proof. induction n as [|n rec].... Qed.
(4) Theorem adder_correct: forall n
    (X Y: data (S n)) c,
(5) |[X]| + |[Y]| + |c| =
    |(LAdder·adder X Y c)|.
(6) Proof. intros; rewrite LA_CLA_eq.
    apply CLAdder.adder_correct. Qed.
```

Reference [18] proposed an extension of Ling's adder by the following equations:

$$D_{j:k} = G_{j:k} + P_{j:k} = G_{j:k+1} + P_{j:k}, \tag{9}$$

$$B_{j:k} = g_j + g + j - 1 + \cdots + g_k, \tag{10}$$

$$G_{j:i} = D_{j:k} \wedge \left(B_{j:k} + G_{k-1:i}\right), \tag{11}$$

where $G_{i:j}$ and $P_{i:j}$ are group propagated and generated carries which are defined later in Section 5. Equation (11) is also proved in this work.

5. Parallel Prefix Adder

CLA improves RCA by computing all the carries in advance as shown in (4). However, large fan-in and fan-out will be caused if all the carries c_i are computed this way especially when i is large. Parallel Prefix Adder (PPA) avoids this by the idea of divide-and-conquer, which provides an efficient way to compute all the parallel carries. Basic definitions are as follows:

$$c_{i+1} = G_{i:j} \vee \left(P_{i:j} \wedge c_j\right) \quad \left(j \leq i\right), \tag{12}$$

$$s_i = c_i \oplus P_i, \tag{13}$$

$$P_{i:j} = \begin{cases} P_i & i = j \\ P_i \wedge P_{i-1:j} & i > j, \end{cases} \tag{14}$$

$$G_{i:j} = \begin{cases} G_i & i = j \\ G_i \vee \left(P_i \wedge G_{i-1:j}\right) & i > j. \end{cases} \tag{15}$$

Due to the similarity between (14) and (15), only the formalization of (15) is shown as follows. An auxiliary function, defined recursively on the difference of i and j, is reluctantly introduced to define it in Coq.

```
(1) Definition GpG_rec n (gp gg: data (S n))
    (d i: nat): bit.
(2) revert i; induction d as [|d rec];
    intros i.
(3) + exact (nth (n-i) gg).
(4) + exact ((nth (n-i) gg) ∨
    ((nth (n-i) gp) ∧ (rec (pred i)))).
(5) Defined.
(6) Definition GpG n (X Y: data (S n)) i j:=
(7) GpG_rec X Y (i-j) i.
```

In line 1, the parameters *gp* and *gg* stand for the propagated and generated carry vectors. Another parameter d is the difference of i and j. Function *n*th k v returns the kth element of v from the leftmost bit indexed 0. *pred n* computes the predecessor of n.

To compute all the carries parallel in advance, the carry c_{i+1} should not depend on any c_k, where $i \geq k > 0$, except c_0 which is the overall carry-in. Therefore, carries of PPA are computed according to a variation of (12) as follows:

$$c_{i+1} = G_{i:0} \vee \left(P_{i:0} \wedge c_0\right), \tag{16}$$

and different PPAs employ different parallel prefix methods to compute the group carries $G_{i:0}$ and $P_{i:0}$, for all $n \geq i \geq 0$, for the sake of high performance. To capture various PPAs in a uniform framework, an abstract module, which abstractly describes this method as *groups*, is employed as follows:

```
(1) Module Type GroupCarries.
(2) Parameter groups: forall n, data2
    (S n) -> data2 (S n).
(3) Axiom groups_correct: forall
    n (X Y: data (S n)),
(4) groups (P X Y, G X Y) =
    correct_groups (P X Y, G X Y).
(5) End GroupCarries.
```

*data*2 *n*, in line 3, is the dependent type of a tuple of vectors whose lengths are both n. Therefore, the parameter of *groups* stands for vectors of propagated and generated carries as shown in line 6. *groups_correct* in line 4 is the assumption that the *groups* function is correct. The correctness is represented as an extensional equality of another correct function and itself. In line 7, *correct_groups* is the correct function to compute the groups carries according to (14) and (15). Its correctness holds by, first, computing all the carries *correct_carries* according to this function and then proving that *correct_carries* are equivalent to the carries of CLA.

```
(1) Definition correct_carries (n: nat)
    (c_in: bool)
(2) (X Y: data (S n)): hyb (S n).
(3) set (PXY:= P X Y).
(4) set (GXY:= G X Y).
(5) set (bvGp:= correct_groups (PXY, GXY)).
(6) set (all_c:= shift_map c_in bvGp).
(7) exact (all_c ▷, all_c ◁).
(8) Defined.
(9) Lemma carries_correct: forall n
    (X Y: data (S n)) c_in,
(10) correct_carries c_in X Y =
    CLAdder.carries (P X Y) (G X Y) c_in.
```

shift_map, in line 2, is a compositional operation first iterating Equation (16) on all the $G_{i:0}$ and $P_{i:0}$ which are stored in the vectors of the first and projection of *bvGp* and then shifting the overall carry-in c_0 to get all the carries. Consider that the computation of $G_{i:j}$ depends on the subgroups of the group propagated carries $P_{i:m}$, the *fundamental carry operator* "$\circ$" as in [19] is used to compute the group propagated and generated carries simultaneously in function *correct_groups* and should be used in all implementations of function *groups*. Consider

$$(P, G) \circ \left(P', G'\right) = \left(P \wedge P', G \vee \left(P \wedge G'\right)\right). \tag{17}$$

Function *correct_groups* can be taken as an instance of *groups* function and is only one particular implementation of *groups*, which is verified. There are many other implementations of the *groups* function based on the following lemmas which are proved by induction on the difference between i and m, using (14) and (15):

$$\begin{aligned} P_{i:j} &= P_{i:m} \wedge P_{m-1:j} \quad \left(j < m \leq i\right), \\ G_{i:j} &= G_{i:m} \vee \left(P_{i:m} \wedge G_{m-1:j}\right) \quad \left(j < m \leq i\right). \end{aligned} \tag{18}$$

Equation (18) can be rewritten using $\circ$ operator in one equation. For all $j < m \leq i$,

$$\left(P_{i:j}, G_{i:j}\right) = \left(P_{i:m}, P_{m-1:j}\right) \circ \left(G_{i:m}, G_{m-1:j}\right). \tag{19}$$

Equation (19) shows clearly that any group of group carries can be computed by its concatenation (or even overlapped) subgroups. And the proper dividing and conquering of the bits of input addends can implement *groups* function with high performance. PPA is such a family of adders differing only in the computation of the *groups* function; thus, a general PPA can be formalized and parameterized by module *GroupCarries*.

```
(1) Module PPAdder (Import M: GroupCarries)
    <: GenAdder.
```

```
(2) Definition adder n (X Y: data (S n))
    (c_in: bit): (hyb (S n)).
(3) set (GT0:= groups ((P X Y), (G X Y)).
(4) set (all_carries:= shift_map c_in (GT0_1)
    (GT0_2)).
(5) set (sum:= PC ⊕ (all_carries ◁)).
(6) exact (all_carries ▷, sum).
(7) Defined.
(8) Theorem adder_correct: forall n
    (X Y: data (S n)) c_in,
(9) |[X]| + |[Y]| + |c_in| = |(adder X Y c_in)|.
(10) Proof.
(11) intros n X Y c_in; unfold adder.
(12) rewrite CLAdder.adder_correct.
(13) unfold CLAdder.adder.
(14) rewrite <- carries_correct.
(15) unfold correct_carries.
(16) rewrite groups_correct; trivial.
(17) Qed.
(18) End PPAdder.
```

Line 5, uses the abstract function *groups* from the parameterized module *GroupCarries* to compute all the group carries in advance. *shift_map* function in line 7 implements the operation in (16). Lines 6 and 8 compute all the carries and the sum.

The correctness of PPA is proved based on the assumption *groups_correct* in the abstract parameterized module *GroupCarries*. Line 15 reformats the left part of the equation to the result of what CLA computes. Line 17 uses the result that the carries of CLA are identical to the carries computed by (14), (15), and (16). Finally, the assumption *groups_correct* is used to prove that *groups* computes the same result as (14) and (15) do.

The rest of this section will show, by the example of Kogge-Stone addition algorithm, how this general PPA applies to some specific ones. The algorithm formalized following [20], in which the algorithm is proposed. Other implementations of PPA can be formalized and verified similarly.

```
(1) Module Kogge_Stone <: GroupCarries.
(2) Fixpoint KS_PG_rec (n m: nat)
    (bvPG: data2 (S n)): (data2 (S n)):=
(3) match m with
(4) | O => bvPG
(5) | S m'=> let recur:= (KS_PG_rec m'bvPG)
    in
(6) data2_op1 recur (shiftin_group
    (power2 m')recur)
(7) end.
(8) Definition groups n (PG: data2 (S n)):=
(9) KS_PG_rec (S (log2 (S n))) PG.
(10) Theorem groups_correct: forall n
    (X Y: data (S n)),
(11) groups (P X Y, G X Y) = correct_groups
    (P X Y, G X Y).
(12) End Kogge_Stone_Group_Carry.
```

Lines 2 to 10 describe the main function to define the Kogge-Stone implementation of the *groups* function. m is a simple counter to indicate how many *stages* are needed and which should the logarithm of the data-width be. When initializing, the input *bvPG* stands for two vectors of the propagated and generated carries, respectively, for example, $P_i = P_{i:i}$ and $G_i = G_{i:i}$, for all $n \geq i \geq 0$. At any stage m, this function computes the group carries of maximum length 2^m. A 8-bit kogge-stone adder is taken as an example to illustrate this procedure. At stage 2 ($m' = 2$), the group carries of maximum length 4 has been computed according to line 8:

$$\begin{aligned} recur := \big(&\left[P_{7:4}, P_{6:3}, P_{5:2}, P_{4:1}, P_{3:0}, P_{2:0}, P_{1:0}, P_0\right], \\ &\left[G_{7:4}, G_{6:3}, G_{5:2}, G_{4:1}, G_{3:0}, G_{2:0}, G_{1:0}, G_0\right]\big). \end{aligned} \tag{20}$$

At the next stage ($m = 3$), as in lines 8 and 9, firstly *shiftin_group* function shifts both vectors in the tuple simultaneously $2^{m'}$ times with *true* and *false*, respectively. The result is represented by $recur'$:

$$\begin{aligned} recur' := \big(&\left[P_{3:0}, P_{2:0}, P_{1:0}, P_0, \text{true}, \text{true}, \text{true}, \text{true}\right], \\ &\left[G_{3:0}, G_{2:0}, G_{1:0}, G_0, \text{false}, \text{false}, \text{false}, \text{false}\right]\big). \end{aligned} \tag{21}$$

Secondly, *date2_op1* executes the fundamental operator in (17) with two operands *recur* and $recur'$, and the result is

$$\begin{aligned} &\big(\left[P_{7:0}, P_{6:0}, P_{5:0}, P_{4:0}, P_{3:0}, P_{2:0}, P_{1:0}, P_0\right], \\ &\quad\left[G_{7:0}, G_{6:0}, G_{5:0}, G_{4:0}, G_{3:0}, G_{2:0}, G_{1:0}, G_0\right]\big). \end{aligned} \tag{22}$$

In line 11, *groups* function specifies that the stages needed are $\log_2(n+1)$, where n is the data-width.

The correctness theorem cannot be proved by induction on the data-width as normal, because Kogge-Stone implementation of *groups* function recurses on the stages, not the data-width as shown in the definition of *KS_PG_rec*.

Noticing that, in the theorem, the result of each side of equation is a tuple of vectors, the equality holds if and only if the corresponding elements are identical pairwise.

```
(1) Lemma data_eq_nth_eq_data2: forall n
    (gx gy: (data2 (S n))),
(2) (forall k, k < S n ->
(3) (nth k (fst gx), nth k (snd gx)) =
```

```
(4) (nth k (fst gy), nth k (snd gy))) <->
(5) gx = gy.
```

However, the result of *KS_PG_rec* changes with the stage m, the first thing to prove is an invariant of m stating how this function approaches the result of *correct_group* function gradually with the increasing of the stages. Suppose, without loss of generality, that

$$\begin{aligned} correct\_groups\,(X,Y) &:= \left(\left[P_{n:0},\ldots,P_0\right],\left[G_{n:0},\ldots,G_0\right]\right),\\ KS\_PG\_rec\,(m,Z) &:= \left(\left[P^m_{n:0},\ldots,P^m_0\right],\left[G^m_{n:0},\ldots,G^m_0\right]\right), \end{aligned} \tag{23}$$

for all m, $Z = (X, Y)$ and $n > 0$; then, for all $0 \le k \le n$,

$$\left(P^m_{k:0}, G^m_{k:0}\right) = \begin{cases} \left(P_{k:0}, G_{k:0}\right) & k < 2^m \\ \left(P_{k:(k+1-2^m)}, G_{k:(k+1-2^m)}\right) & k \ge 2^m. \end{cases} \tag{24}$$

With this invariant, the existence of the fixed points can be proved secondly, and the least fixed point should be $\log_2(n+1)$. For all $m \ge \log_2(n+1)$and $k \le n < 2^m$, $(P^m_{k:0}, G^m_{k:0}) = (P^{m+1}_{k:0}, G^{m+1}_{k:0}) = (P_{k:0}, G_{k:0})$. Function *groups*, which iterates *KS_PG_rec* function $\log_2(n+1)$ stages, computes the same result as function *correct_groups*, which is the correctness theorem. The whole proof of this theorem has been carried out in COQ, although they are expressed in an intuitive way here for better understandings of the readers.

Kogge-Stone adder can be combined by the general module of PPA and this specific module of Kogge-Stone methods to compute all the group carries, which provides not only the computation method but also the correctness proof.

```
(1) Module Kogge_Stone <: GenAdder:=
(2) PPAdder Kogge_Stone.
```

6. Conclusion and Future Work

In this work, we proposed a holistic methodology to formalize and verify primary adders (RCA, CLA, LA, and PPA) in theorem prover Coq. They are formalized using dependent types, higher-order recursion and module systems in order to provide fidelity, scalability, and modularization.

In particular, PPA is a family of adders sharing the same structure, only differing in the methods of parallel prefix computing. We provide a novel way to describe the general PPA and show how to use this general module to verify a specific PPA, exemplified by Kogge-Stone adder.

Other advanced arithmetic designs can be verified reusing the formalizations and verifications of this work in a combinational way, as we describe by the example of carry select adders.

All the work has been carried out in Coq. The whole development contains around 2,000 lines of Coq scripts. This number of scripts is only about one third of [12], which is another work dedicated to verify additional designs in Coq. This work used lesser scripts but verified more addition designs than [12].

This work can be continued in two directions. Advanced arithmetic designs, such as IBM POWER6, can be cumulately verified based on these verified adders. Since formalization in a constructive way is to have clear correspondence to gate-level descriptions of circuits, HDL codes can be generated from the verified designs, which may provide an alternative way for designing the correct arithmetic implementations.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work has been supported by the National Science Foundation of China Grant 61272002, the Tsinghua National Laboratory for Information Science and Technology (TNList) Cross-discipline Foundation 2011-9, the Major Research plan of the National Natural Science Foundation of China Grant 91218302, and the National Basic Research Program of China (973 Program) Grant 2010CB328003.

References

[1] W. A. Hunt Jr. and B. C. Brock, "The verification of a bit-slice ALU," in *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, M. Leeser and G. Brown, Eds., vol. 408 of *Lecture Notes in Computer Science*, pp. 282–306, Springer, New York, NY, USA, 1990.

[2] D. Borrione, L. Pierre, and A. Salem, "Formal verification of VHDL descriptions in the prevail environment," *IEEE Design & Test of Computers*, vol. 9, no. 2, pp. 42–56, 1992.

[3] A. Camilleri, M. Gordon, and T. Melham, *Hardware Verification Using Higher-Order Logic*, Computer Laboratory, University of Cambridge, Cambridge, UK, 1986.

[4] P. Curzon, "Experiences formally verifying a network component," in *Proceedings of the 9th Annual Conference on Computer Assurance (COMPASS '94)*, Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security, pp. 183–193, Gaithersburg, Md, USA, July 1994.

[5] C. Paulin-Mohring, "Circuits as streams in Coq: verification of a sequential multiplier," in *Types for Proofs and Programs*, S. Berardi and M. Coppo, Eds., vol. 1158 of *Lecture Notes in Computer Science*, pp. 216–230, Springer, Berlin, Germany, 1996.

[6] S. Coupet-Grimal and L. Jakubiec, "Certifying circuits in type theory," *Formal Aspects of Computing*, vol. 16, no. 4, pp. 352–373, 2004.

[7] The Coq Development Team, "The Coq proof assistant, reference manual," Tech. Rep. version 8.4, INRIA, Roquencourt, France, 2012.

[8] T. Braibant, "Coquet: a coq library for verifying hardware," in *Certified Programs and Proofs*, J.-P. Jouannaud and Z. Shao, Eds., vol. 7086 of *Lecture Notes in Computer Science*, pp. 330–345, Springer, Berlin, Germany, 2011.

[9] G. J. Milne, *Formal Specification and Verification of Digital Systems*, McGraw-Hill, New York, NY, USA, 1993.

[10] J. T. O'Donnell and G. Rünger, "Functional pearl derivation of a logarithmic time carry lookahead addition circuit," *Journal of Functional Programming*, vol. 14, no. 6, pp. 697–713, 2004.

[11] F. Liu, Q. Tan, and G. Chen, "Formal proof of prefix adders," *Mathematical and Computer Modelling*, vol. 52, no. 1-2, pp. 191–199, 2010.

[12] G. Chen, "Formalization of a parameterized parallel adder within the Coq theorem prover," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 1, pp. 149–153, 2010.

[13] D. Kapur and M. Subramaniam, "Mechanical verification of adder circuits using rewrite rule laboratory," *Formal Methods in System Design*, vol. 13, no. 2, pp. 127–158, 1998.

[14] R. Hinze, "An algebra of scans," in *Mathematics of Program Construction*, D. Kozen, Ed., vol. 3125 of *Lecture Notes in Computer Science*, pp. 186–210, Springer, Berlin, Germany, 2004.

[15] B. Barras, J. P. Jouannaud, P. Y. Strub, and Q. Wang, "CoQMTU: a higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory," in *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS '11)*, pp. 143–151, Ontario, Canada, 2011.

[16] Q. Wang and B. Barras, "Semantics of intensional type theory extended with decidable equational theories," in *Computer Science Logic (CSL '13)*, S. R. D. Rocca, Ed., vol. 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 653–667, Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013.

[17] H. Ling, "High-speed binary adder," *IBM Journal of Research and Development*, vol. 25, no. 3, pp. 156–166, 1981.

[18] R. Jackson and S. Talwar, "High speed binary addition," in *Proceedings of the Conference Record of the 38th Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1350–1353, Asilomar, Calif, USA, November 2004.

[19] I. Koren, *Computer Arithmetic Algorithms*, Universities Press, Hyderabad, India, 2002.

[20] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. 2, no. 8, pp. 786–793, 1973.