

Functional Verification of High Performance Adders in COQ

Qian Wang¹, Xiaoyu Song³, Ming Gu², and Jianguang Sun²

¹ School of Software, Tsinghua University, Beijing, 100084, China
q-w04@mails.tsinghua.edu.cn

² School of Software, Tsinghua University, Beijing, 100084, China
{guming, sunjg}@tsinghua.edu.cn

³ Department of ECE, Portland State University, Portland, OR 97207, USA
song@ece.pdx.edu

Abstract. 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 challenge topic which has lasted for several decades. Testing and simulation, as the traditional methods, have won good reputation and 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 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, but supposed intricate to carry out because 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 a 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 sub-components. This methodology provides an 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.

- Scalability : The formalization of an adder is parameterized by a natural number (named *length*) and the correctness proof applies to any length.
- Modularization : Various verified adders are encapsulated as instances of an abstract module, which provides an 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 the their detailed implementations.
- 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 web page⁴. 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 are not at the fingertips. In particular, the formalization and verification of the Ling adder, cannot be found in any literature. [8] proves the correctness of RCA by formalizing adders with dependent types in Coq. [9] proves the correctness of RCA by the higher order logic with a reusable library for formalizing circuits. [2] verifies RCA written in VHDL as well as other circuits by the higher order logic. [10] develops semi-formal correctness proof of CLA or PPA. [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

⁴ <https://github.com/superwalter/VeriAdder/tree/master/V2>

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 previous 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 An Unified Proof Structure

Basically, the methodology answers four questions:

- how to formalize the related data types,
- which method is used to formalize an adder,
- what should be proved, and
- how to organize formalizations and verifications for different adders.

These questions are answered by an 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 can it 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 . *hyb* 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 output 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. An 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 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 Equation 1 until c_0 which is overall input bit as shown by the following example.

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

This process as well as definitions of P , G are formalized as:

```

1  Definition P n (X Y : data n) := X  $\oplus$  Y.
2  Definition G n (X Y : data n) := X  $\wedge$  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  $\triangleright$ ) (X  $\triangleright$ ) cin, [cin]).

```

```

6 + set (recs := rec (X <) (Y <)).
7 exact (bandor (Y >) (X >) (recs1), (recs1)⋈(recs2)).
8 Defined.

```

\oplus , \wedge in line 1, 2 and \vee used later are extensions of logical boolean operations \oplus , \wedge and \vee , 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 \triangleright operators in line 5 returns the leftmost element of a vector. Correspondingly, the \triangleleft 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 \bowtie 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 (cc1, (P X Y)  $\oplus$  (cc2)).
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 use this methodology for the verification of adders.

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:

```

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 CLA 3, 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.

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 input carry in very late steps, 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.
```

Line 2 to 10 define CSA. Two adders compute the sum and the carry-out with respect to carry-in *true* and *false* in line 4 and 5 respectively. The multiplexer chooses the real sum and carry-out according to the actual carry-in in line 6 and 7, which is the first time input carry is needed. *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 is quite complex due to the problem with the dependent types as described in [15] and [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 rest 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 a argument. The initial values of this recursive function is 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.

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 non-dependent type *list* as in [12]. Both [6] and [8] have explanations why dependent type is more proper for the verification of adders. Generally speaking, non-dependent 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 are 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 less 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 Equations 3 and 4 respectively as follows.

$$k_i = a_i \wedge b_i \quad (3)$$

$$T_i = a_i \vee b_i \quad (4)$$

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 Equation 5b.

$$H_i = \begin{cases} k_i \vee (H_{i-1} \wedge T_{i-1}) & i > 0 \\ k_i \vee c_{in} & i = 0 \end{cases} \quad (5a)$$

$$(5b)$$

Without this case, the default value of H_{-1} and T_{-1} are both *false*, and it is equivalent to our definition assuming c_{in} is always *false*. More intuitively, that algorithm does not consider the carry-in to the least significant bit, which

restricts it only suitable for some industrial uses, 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.

$$s_i = \begin{cases} (H_i \oplus T_i) \vee (k_i \wedge H_{i-1} \wedge T_{i-1}) & i > 0 \\ (H_i \oplus T_i) \vee (k_i \wedge c_{in}) & i = 0 \end{cases} \quad (6a) \quad (6b)$$

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 i_{th} pseudo carry of H , the i_{th} bit of K and the $(i-1)_{th}$ bit of T is needed. Therefore, the two parameters of H stands 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 Equation 6a and 6b 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 i_{th} bit of sum depends on the $(i-1)_{th}$ bit of H and T , they are shifted in line 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 $(TXY \triangleright) \wedge (Hc \triangleright)$ which is equivalent to c_{out} as shown in Equation 7.

$$c_i = H_{i-1} \wedge T_{i-1}, i \geq 0 \quad (7)$$

The formalization of Equation 7 is complicated, but the proof is trivial by induction and case analysis. The correctness of LA follows by proving an lemma stating that the outputs of CLA and LA are the same with regards to arbitrary same inputs. This lemma is proved by induction with the result of Equation 7.

```

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.

```

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

$$D_{j:k} = G_{j:k} + P_{j:k} = G_{j:k+1} + P_{j:k} \quad (8)$$

$$B_{j:k} = g_j + g + j - 1 + \dots + g_k \quad (9)$$

$$G_{j:i} = D_{j:k} \wedge (B_{j:k} + G_{k-1:i}) \quad (10)$$

where $G_{i:j}$ and $P_{i:j}$ are group propagated and generated carries which are defined later in the Section 5. Equation 10 is also proved in this work.

5 Parallel Prefix Adder

CLA improves RCA by computing all the carries in advance as shown in Equation 3. 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 divide-and-conquer the bits of input addends, which provides an efficient way to compute all the carries parallel. Basic definitions are as follow.

$$c_{i+1} = G_{i:j} \vee (P_{i:j} \wedge c_j) \quad (j \leq i) \quad (11)$$

$$s_i = c_i \oplus P_i \quad (12)$$

$$P_{i:j} = \begin{cases} P_i & i = j \\ P_i \wedge P_{i-1:j} & i > j \end{cases} \quad (13)$$

$$G_{i:j} = \begin{cases} G_i & i = j \\ G_i \vee (P_i \wedge G_{i-1:j}) & i > j \end{cases} \quad (14)$$

Due to the similarity of the Equation 13 and 14, only the formalization of Equation 14 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) gp) ∨ ((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 $nth\ k\ v$ returns the k_{th} element of v from leftmost bit indexed 0. $pred\ n$ is a function returns $n - 1$.

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 Equation 11 as follows.

$$c_{i+1} = G_{i:0} \vee (P_{i:0} \wedge c_0) \quad (15)$$

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 an 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.

```

$data2\ n$, in line 3, is the dependent type of a tuple of vectors whose length 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 Equations 13 and 14. Its correctness holds by first computing all the carries *correct_carries* according to this function, and then proving *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,

```

shift_map, in line 2, is a compositional operation first iterating Equation 15 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* 'o' 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*.

$$(P, G) \circ (P', G') = (P \wedge P', G \vee (P \wedge G')) \quad (16)$$

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 of i and m , using Equation 13 and 14.

$$P_{i:j} = P_{i:m} \wedge P_{m-1:j} \quad (j < m \leq i) \quad (17)$$

$$G_{i:j} = G_{i:m} \vee (P_{i:m} \wedge G_{m-1:j}) \quad (j < m \leq i) \quad (18)$$

Equation 17 and 18 can be rewritten using \circ operator in one equation. For all $j < m \leq i$,

$$(P_{i:j}, G_{i:j}) = (P_{i:m}, P_{m-1:j}) \circ (G_{i:m}, G_{m-1:j}) \quad (19)$$

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 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 (GT01) (GT02)).
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 Equation 15. line 6 and 8 compute all the carries and the sum.

The correctness of a 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 Equation 13, 14 and 15. Finally, the assumption *groups_correct* is used to prove that *groups* computes the same result as Equation 13 and 14 does.

The rest of this section will show, by the example of Kogge-Stone addition algorithm, that how this general PPA applies to some specific one. The algorithm formalized follows [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  | 0 => bvPG
5  | S m' => let recur := (KS_PG_rec m' bvPG) in
6  data2_op1 recur (shiftn_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* needed, which should be the logarithm of the data-width. When initializing, the input *bvPG* stands for two vectors of the propagated and generated carries respectively, e.g. $P_i = P_{i:i}$ 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 . For an 8-bit Kogge-Stone adder, suppose at stage 2 ($m' = 2$), all groups carries with maximum length 4 has been computed as in line 8:

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

At the next stage ($m = 3$), as in lines 8 and 9, firstly *shiftn_group* function shifts both vectors in the tuple simultaneously $2^{m'}$ times with *true* and *false*

respectively. The result is represented by $recur'$:

$$recur' := ([P_{3:0}, P_{2:0}, P_{1:0}, P_0, true, true, true, true], \\ [G_{3:0}, G_{2:0}, G_{1:0}, G_0, false, false, false, false])$$

Secondly, $date2_op1$ executes the fundamental operator in Equation 16 with two operands $recur$ and $recur'$, and the result is :

$$([P_{7:0}, P_{6:0}, P_{5:0}, P_{4:0}, P_{3:0}, P_{2:0}, P_{1:0}, P_0], \\ [G_{7:0}, G_{6:0}, G_{5:0}, G_{4:0}, G_{3:0}, G_{2:0}, G_{1:0}, G_0])$$

In line 11, $groups$ function specifies that the stages needed are $\log_2(n+1)$, where n is the date-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 iff 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

$$correct_groups(X, Y) := ([P_{n:0}, \dots, P_0], [G_{n:0}, \dots, G_0]) \\ KS_PG_rec(m, Z) := ([P_{n:0}^m, \dots, P_0^m], [G_{n:0}^m, \dots, G_0^m])$$

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

$$(P_{k:0}^m, G_{k:0}^m) = \begin{cases} (P_{k:0}, G_{k:0}) & k < 2^m \\ (P_{k:(k+1-2^m)}, G_{k:(k+1-2^m)}) & k \geq 2^m \end{cases}$$

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 $m \geq \log_2(n+1)$, $k \leq n < 2^m$, $(P_{k:0}^m, G_{k:0}^m) = (P_{k:0}^{m+1}, G_{k:0}^{m+1}) = (P_{k:0}, G_{k:0})$. 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. All the proofs of this theorem are carries 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 not only provides 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, 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 addition designs in COQ. This work used less 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 cumulatively verified based on these verified adders. Since formalization in a constructive way to have a clear correspondences 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.

Acknowledgment

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

Conflict of Interests

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

References

1. Hunt, Warren A., J., Brock, B.: The verification of a bit-slice alu. In Leeser, M., Brown, G., eds.: *Hardware Specification, Verification and Synthesis: Mathematical Aspects*. Volume 408 of *Lecture Notes in Computer Science*. Springer New York (1990) 282–306
2. Borrione, D., Pierre, L., Salem, A.: Formal verification of vhdl descriptions in the prevail environment. *Design Test of Computers, IEEE* **9**(2) (1992) 42–56

3. Camilleri, A., Gordon, M., Melham, T.: Hardware verification using higher-order logic. Computer Laboratory, University of Cambridge (1986)
4. Curzon, P.: Experiences formally verifying a network component. In: Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on. (1994) 183–193
5. Paulin-Mohring, C.: Circuits as streams in coq: Verification of a sequential multiplier. In Berardi, S., Coppo, M., eds.: Types for Proofs and Programs. Volume 1158 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1996) 216–230
6. Coupet-Grimal, S., Jakubiec, L.: Certifying circuits in type theory. Formal Aspects of Computing **16**(4) (2004) 352–373
7. The Coq Development Team: The Coq Proof Assistant, Reference Manual, Version 8.4. Technical report, INRIA, Roquencourt, France (2012)
8. Braibant, T.: Coquet: a coq library for verifying hardware. CoRR **abs/1108.4253** (2011)
9. Milne, G.J.: Formal Specification and Verification of Digital Systems. McGraw-Hill, Inc., New York, NY, USA (1993)
10. O'DONNELL, J.T., RINGER, G.: Functional pearl derivation of a logarithmic time carry lookahead addition circuit. Journal of Functional Programming **14** (10 2004) 697–713
11. Liu, F., Tan, Q., Chen, G.: Formal proof of prefix adders. Mathematical and Computer Modelling **52**(12) (2010) 191 – 199
12. Chen, G.: Formalization of a parameterized parallel adder within the coq theorem prover. Trans. Comp.-Aided Des. Integ. Cir. Sys. **29**(1) (January 2010) 149–153
13. Kapur, D., Subramaniam, M.: Mechanical verification of adder circuits using rewrite rule laboratory. Formal Methods in System Design **13**(2) (1998) 127–158
14. Hinze, R.: An algebra of scans. In Kozen, D., ed.: Mathematics of Program Construction. Volume 3125 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 186–210
15. Barras, B., Jouannaud, J.P., Strub, P.Y., Wang, Q.: Coqmtu: A higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory. In: Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on. (2011) 143–151
16. Wang, Q., Barras, B.: Semantics of Intensional Type Theory extended with Decidable Equational Theories. In Rocca, S.R.D., ed.: Computer Science Logic 2013 (CSL 2013). Volume 23 of Leibniz International Proceedings in Informatics (LIPIcs)., Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2013) 653–667
17. Ling, H.: High-speed binary adder. IBM Journal of Research and Development **25**(3) (1981) 156–166
18. Jackson, R., Talwar, S.: High speed binary addition. In: Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on. Volume 2. (2004) 1350–1353 Vol.2
19. Koren, I.: Computer arithmetic algorithms. Universities Press (2002)
20. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. Computers, IEEE Transactions on **C-22**(8) (1973) 786–793