Formal Verification of Tree-Structured Carry-Lookahead Adders

Sae Hwan Kim
Department of Electrical Engineering
and Computer Science
Syracuse University, Syracuse, NY 13244
sakim@syr.edu

Shiu-Kai Chin
Department of Electrical Engineering
and Computer Science
Syracuse University, Syracuse, NY 13244
skchin@syr.edu

Abstract

Quad trees - trees with four branches, are used to abstractly describe tree-structured carry-lookahead adders using 4-bit components. The specification and implementation descriptions are parameterized and describe tree-structured adders having arbitrarily large inputs and outputs. The descriptions are formally verified using the HOL theorem prover.

1. Introduction

Carry-lookahead adders are widely used due to their superior performance over ripple adders. The expressive power of higher-order logic makes it possible to define a wide variety of data types and to prove theorems that concisely and abstractly state the properties of these types.

Implementations and specifications are described parametrically using higher-order logic and types. These parametric descriptions are proved correct once and subsequently are specialized to specific values, e.g., to specific word lengths.

Carry-lookahead adders are explained in Section 2. The quad tree type *qtree* is defined in Section 3. Using *qtree*, a parameterized implementation description of tree-structured carry-lookahead adders is defined. Its correctness is stated in Section 4. The conclusions are in Section 5.

2. Carry-Lookahead Adders

Carry-lookahead adder (CLA) circuits are constructed using carry-lookahead adders and carry-lookahead generators (CLG). Figure 1 shows a 16-bit adder as an example. Each 4-bit CLA produces a 4-bit slice of the sum. The CLG generates the carry inputs for the three most significant carry-lookahead adders.

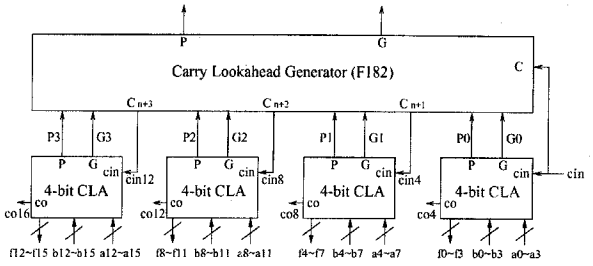

Figure 1. A 16-Bit Carry-Lookahead Adder

The structure of the adder in Figure 1 is a *quad tree*. The carry-lookahead generator is the *node* of the quad tree. Each 4-bit carry-lookahead adder is a *leaf* of the quad tree.

3. Tree-Structured Adders

3.1. Describing Implementations Using Quad Trees

Quad trees belong to the recursive type *qtree* described informally by:

qtree ::= **LEAF** | **NODE** *qtree qtree qtree qtree*

Quad trees are either a **LEAF** or a **NODE** connected to four other *qtrees*.

Tree-structured adders using 4-bit carry-lookahead circuits are described abstractly using quad trees. Each **NODE** corresponds to a carry-lookahead generator. Each **LEAF** corresponds to a 4-bit carry-lookahead adder. For example, the 40-bit adder in Figure 2 is described by:

NODE (NODE LEAF LEAF LEAF (NODE LEAF LEAF LEAF LEAF)) LEAF LEAF LEAF

0-7695-0104-4/99 $10.00 © 1999 IEEE

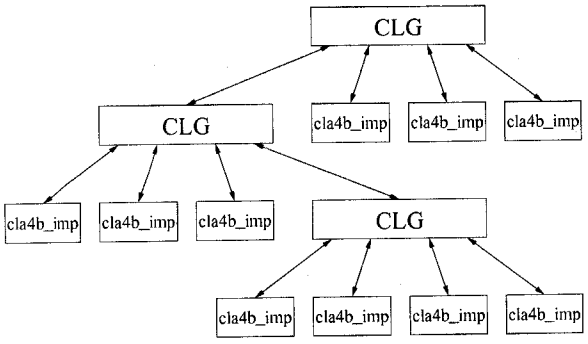

Figure 2. 40-Bit Tree-Structured Adder

3.2. Hierarchical Hardware Descriptions

The abstract quad tree structural descriptions are converted into hierarchical hardware descriptions with input and output signals, 4-bit carry-lookahead adders `cla4b_imp`, and carry-lookahead generators `CLG`, by the predicate `tree_cla_imp`. The arguments of `tree_cla_imp` are a quad tree t, carry input *cin*, adder inputs a and b, output f, carry output *co*, and carry generate and propagate signals G and P.

In the base case, the quad tree argument t is `LEAF`. `tree_cla_imp` describes a 4-bit carry-lookahead adder, `cla4b_imp`:

```
⊢def  ∀cin a b f co G P.
        tree_cla_imp LEAF cin a b f co G P =
        (∃a0 a1 a2 a3 b0 b1 b2 b3 f0 f1 f2 f3.
          (a = WORD [a3; a2; a1; a0]) ∧
          (b = WORD [b3; b2; b1; b0]) ∧
          (f = WORD [f3; f2; f1; f0]) ∧
          cla4b_imp cin a0 a1 a2 a3 b0 b1 b2 b3
                    f0 f1 f2 f3 co G P)
```

For the recursive case where the quad tree argument t is of the form *NODE t3 t2 t1 t0*, `tree_cla_imp` connects four subtrees *t3, t2, t1*, and *t0* together using carry-lookahead generator `CLG`. Subtree input and output words *a3, a2, a1, a0*, and *b3, b2, b1, b0* are concatenated together using `WCAT` to form the input and output words a and b of the whole adder. The output f is obtained by concatenating subtree outputs *f3, f2, f1*, and *f0*. The detailed structure of subtrees *t3, t2, t1*, and *t0* is obtained by recursively applying `tree_cla_imp` to each subtree.

```
⊢def  ∀t3 t2 t1 t0 cin a b f co G P.
        tree_cla_imp (NODE t3 t2 t1 t0) cin a b f co G P =
        (∃a0 a1 a2 a3 b0 b1 b2 b3 f0 f1 f2 f3 cin4 cin8
          cin12 co4 co8 co12 G0 P0 G1 P1 G2 P2 G3 P3.
          (a = WCAT (a3,WCAT (a2,WCAT (a1,a0)))) ∧
          (b = WCAT (b3,WCAT (b2,WCAT (b1,b0)))) ∧
          (f = WCAT (f3,WCAT (f2,WCAT (f1,f0)))) ∧
          tree_cla_imp t0 cin a0 b0 f0 co4 G0 P0 ∧
          tree_cla_imp t1 cin4 a1 b1 f1 co8 G1 P1 ∧
          tree_cla_imp t2 cin8 a2 b2 f2 co12 G2 P2 ∧
          tree_cla_imp t3 cin12 a3 b3 f3 co G3 P3 ∧
          CLG cin G0 P0 G1 P1 G2 P2 G3 P3 cin4 cin8 cin12
              G P)
```

4. Correctness

Informally, the behavioral specification is the sum of the values of the inputs *cin*, a, and b equals the value of the outputs f and *co* summed together. This is formally specified by `add_spec` where `BNVAL`, `BV`, `EXP`, and `WORDLEN` denote the binary word value, bit value, exponent, and word length functions, respectively.

```
⊢def  ∀cin a b f co.
        add_spec cin a b f co =
        BNVAL a + BNVAL b + BV cin =
        BNVAL f + BV co * 2 EXP WORDLEN f
```

The correctness of `tree_cla_imp` with respect to `add_spec` is proved in [2] using the HOL theorem prover [1]. The following correctness theorem states that the behavior of `tree_cla_imp` is completely contained by `add_spec`.

```
⊢  ∀t cin a b f co G P.
     tree_cla_imp t cin a b f co G P ⊃
     add_spec cin a b f co
```

5. Conclusions

Many hardware devices are built using tree-shaped structures. The approach outlined here, using parameterization and *tree* data types for abstract structural descriptions, should be applicable beyond carry-lookahead adders. The advantage of using abstract structural descriptions is the support of hierarchical descriptions that ease the verification of implementations with respect to purely behavioral specifications.

References

[1] M. Gordon. A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI specification, verification and synthesis*. Kluwer, 1987.

[2] S. H. Kim. Formal Verification of Tree-Structured Carry-Lookahead Adders. Master's thesis, Syracuse University, 1999.