Automated Formal Synthesis of Wallace Tree Multipliers

Osman Hasan
Dept. of Electrical & Computer Engineering
Concordia University
Montreal, Quebec, H3G 1M8
Email: o_hasan@ece.concordia.ca

Skander Kort
Dept. of Electrical & Computer Engineering
Concordia University
Montreal, Quebec, H3G 1M8
Email: kort@ece.concordia.ca

***Abstract*— In this paper, we present a formal synthesis methodology that is capable of performing correct synthesis at almost all levels of abstraction and can be adapted to be used for most of the combinational digital circuits irrespective of their size and complexity. The proposed methodology calls for proving the correctness-preserving characteristic for the transformations that are required in the synthesis of a particular digital circuit in a higher-order-logic theorem prover. These correctness-preserving transformation proofs can then be used to automatically verify the correctness of the corresponding synthesis process within the theorem prover in an automated way. For illustration purposes, we present the construction of an automated formal synthesis tool for Wallace Tree multipliers based on our methodology.**

I. Introduction

Due to the increased complexity of digital circuits and their corresponding synthesis algorithms, the *correctness-by-construction* paradigm claimed by most automated synthesis tools cannot be guaranteed. Therefore, in order to ensure correct functionality of the final implementation of digital circuits, a significant portion of the design time is spent in proving the correspondence between the synthesized results and the given specifications using hardware verification techniques. A very promising alternative is to use the Formal Synthesis approach [9], which allows us to formally derive the synthesis results within a formal environment and thus omits the requirement of post-synthesis verification. In formal synthesis, the circuit descriptions are formalized in a mathematical manner and the synthesis process is restricted to logical transformations that preserve the correctness of the original circuit specifications, usually referred to as the correctness-preserving transformations. Therefore, in contrast to conventional synthesis, the correctness of the synthesis procedure is guaranteed to be correct in an implicit manner.

One major limitation of formal synthesis is that end users who perform the actual synthesis need to be familiar with formal semantics and reasoning techniques. Nowadays, designers working in the industry lack expertise in these domains and prefer automated, push button type tools. In this paper, we propose a formal synthesis methodology that tends to bridge this gap. The main idea behind our methodology is to use a theorem prover to verify the correctness-preserving characteristic of a set of synthesis transformations, prior to the actual synthesis process. Any computer aided synthesis process that is composed of this set of synthesis transformations can then be verified in an automated way. Our methodology is quite general and can be used to build specialized formal synthesis based automatic tools for any kind of digital circuit. For illustration purposes, we present the construction details of an automatic synthesis tool specialized for synthesizing Wallace Tree (WT) multipliers in this paper. We have selected the higher-order-logic theorem prover Isabelle/HOL [11] for the verification part and the synthesis tool is developed in C++.

II. Synthesis Methodology

The proposed methodology, illustrated in Fig. 1, consists of two major components; a synthesis and a validation tool. We use a higher-order-logic theorem prover as the validation tool to verify the correctness-preserving characteristic of a set of synthesis transformations, prior to the actual synthesis process. The synthesis tool, which is a specialized software program capable of performing synthesis of a particular digital circuit, has access to this pre-verified set of synthesis transformation along with certain built-in compression and optimization algorithms. It accepts the specification of the digital circuit under consideration and generates a sequence of synthesis transformations, called transformation trace (TT), which can be used to generate the synthesized netlist when applied to the given specification. The synthesis tool then replays this TT to obtain the synthesized netlist and also generates a correctness lemma that checks the correctness-preserving characteristic of the TT for the current synthesis process in higher-order-logic. The correctness lemma serves as a bridge between the synthesis and validation tools and can be automatically verified since the proof process merely consists of checking the fact that all the transformations in the TT have already been verified to be correctness-preserving.

It is important to note that the basis of our methodology is the correctness-preserving characteristic of synthesis transformations. This fact differentiates our methodology from post-synthesis verification where no information about the actual synthesis process is available. The presented synthesis methodology is completely automatic and the end user only needs to supply the circuit specification. The final output is a synthesized netlist in some hardware description language (HDL) accompanied by a formal proof of its correctness.

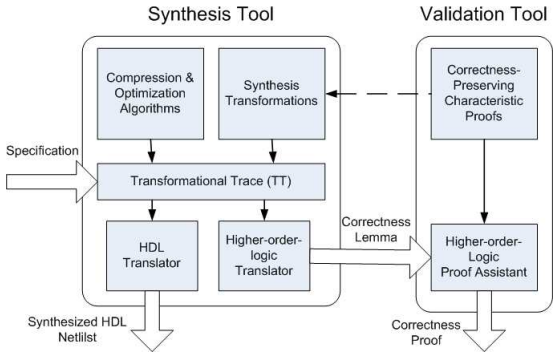

Fig. 1. Synthesis Methodology

III. Wallace Tree Multipliers

The WT multiplier [13] sums up all the bits of the same weights in a merged tree rather than completely adding the partial products

in pairs. Full adder (FA) and Half adder (HA) cells are used to add three or two equally weighted bits respectively to produce two bits: the sum bit with a weight equal to that of the operands and the carry bit with a weight equal to one more than that of the operands. The height of the WT is reduced by a factor of 3:2, whenever a FA is used. The final tree is composed of as many levels of FA and HA cells as are necessary to reduce the height of the tree to 2.

The hardware synthesis process for a WT multiplier mainly consists of two steps. The first step is to arrange the partial product bits as the initial WT structure, as shown in Fig. 2 for the case of a 4x4 multiplier with operands (a_3, a_2, a_1, a_0) and (b_3, b_2, b_1, b_0). Secondly, a series of FA and HA transformations are applied on the WT structure until the tree height is reduced to 2. At this point, any n-bit conventional adder may be used to add the remaining two n-bit rows of the tree to get the final multiplication result.

```
b3a3   b2a3   b1a3   b0a3   b0a2   b0a1   b0a0
       b3a2   b2a2   b1a2   b1a1   b1a0
              b3a1   b2a1   b2a0
                     b3a0
```

Fig. 2. Initial WT structure for a 4x4 Multiplier

In this paper, we present the automated formal synthesis of WT multipliers using the proposed methodology. For correctly synthesized WT multipliers we have to make sure that both FA and HA transformations are correctness-preserving and for the synthesis process to be complete we have to make sure that the final height of the WT structure is reduced to 2. As outlined in Section II, our synthesis methodology is composed of two major components; the synthesis tool and the validation tool. The synthesis tool in the case of WT multiplier synthesis accepts the widths of the operands and provides the synthesized gate level netlist in some HDL along with the correctness lemma for the WT synthesis process. The validation tool contains the formal proofs for the correctness-preserving characteristic of FA and HA transformations and thus automatically proves the correctness lemma generated by the synthesis tool.

IV. Wallace Tree Multiplier Formalization

This section presents the formalization of WT structure and the two synthesis transformation, FA and HA, in Isabelle/HOL. This formalization is required to verify the correctness-preserving characteristics of FA and HA transformations as well as to prove the correctness of the WT synthesis process.

A. Formalization of WT structure

We modeled the WT structure as a list of columns, where each column is a list of bits, by a higher-order-logic function, *wallace_tree*. This function accepts the two multiplication operands as bit-lists and generates the initial WT structure model (Fig 2). The initial WT structure generation process can be divided into two major tasks; generation of partial products and the arrangement of these partial products in the initial WT structure format.

The partial products are generated by two recursive functions. The function *list_and* recursively performs the logical *and* operation between a bit and all the bits of a bit list to form a list of partial products. Whereas, the function *par_prod* accepts two bit-lists (the multiplicand and the multiplier) and recursively calls function *list_and* with the multiplicand and all the bits of the multiplier one by one to generate a two dimensional partial product list or a pre-arranged WT structure as shown for a 4x4 multiplier in Fig 3(a).

The WT structure arrangement process can be subdivided into three steps. Fig. 3 illustrates the arrangement process for a 4x4 WT

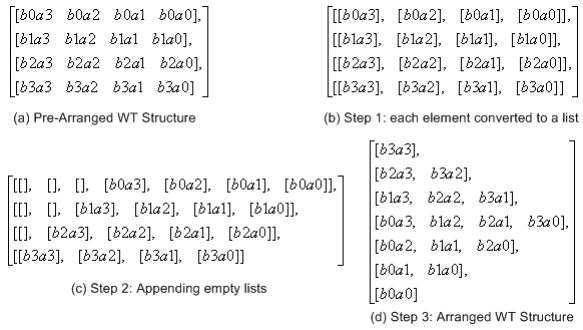

Fig. 3. Initial WT structure generation for a 4x4 Multiplier in Isabelle/HOL

structure and it can be extended to any WT structure. In the first step, a list with only one element is created for each element in the pre-arranged WT structure (Fig 3.b). In the second step, empty lists (Nil []) are appended to each row of the structure obtained after step 2 such that the number of empty lists is equal to the number of remaining rows, e.g., 3 empty lists are appended to the first row of Fig. 3.c. In the third step, the n^{th} elements of each row are concatenated in order to form the arranged WT structure (Fig. 3.d), which represents the initial WT structure, as shown in the case of a 4x4 multiplier in Fig 2, where each partial product list corresponds to a WT column.

We used the integer value of a WT structure to define its semantics as it is a unique characteristics of every WT structure. The conversion from a WT structure to an integer value is defined by two recursive functions. Function *eval_col* computes the integer value of a Wallace tree column. Function *eval_wts* accepts the WT structure and recursively calls the function *eval_col* for all the WT structure columns to obtain the integer value of the whole WT structure

```
eval_col :: WT_col → nat → int
∀ i a as. eval_col Nil i = 0
   eval_col (Cons a as) i =
   ((bitval a) * 2^i) + (eval_col as i)
eval_wts :: WTS → nat → int
∀ i a as. eval_wts Nil i = 0
   eval_wts (Cons a as) i =
   (eval_col a i) + (eval_wts as (Suc i)
```

B. Formalization of WT synthesis transformations

The WT synthesis process involves two types of synthesis transformations; FA and HA, which basically perform bit-level addition on the top 3 and 2 bits of a WT column, respectively. The formalization details of a FA transformation function, *fa_trans*, are as follows:

```
sum_fa :: bit → bit → bit → bit
∀ a b c. sum_fa a b c = a⊕b⊕c
cout_fa :: bit → bit → bit → bit
∀ a b c. cout_fa a b c =(a.b)+(a⊕b).c
fa_msb :: bit list → bit
∀ a. fa_msb a = cout_fa (bv_msb a)
   (bv_msb (tl a)) (bv_msb (tl(tl a)))
fa_lsb :: bit list → bit
∀ a. fa_lsb a = sum_fa (bv_msb a)
   (bv_msb (tl a)) (bv_msb (tl(tl a)))
fa_trans ::
 bit list → bit list → bit list list
∀ as bs. fa_trans as bs = ((fa_lsb as)#
   (tl(tl(tl as))))#((fa_msb as)# bs)#[]
```

The functions *sum_fa* and *cout_fa* implement the bit level full adder *sum* and *carry* bits. The functions *fa_msb* and *fa_lsb* accept a list of bits and return the sum and carry bits obtained by adding the top three bits of this list respectively. The function *fa_trans* accepts two bit lists and adds the top three bits of the first list, and replaces them with their sum bit and appends the carry bit of these three bits on top of the second bit list and returns the concatenation of these two modified bit lists as a two dimensional bit list. The HA transformation, *ha_trans*, is also formalized similarly.

A valid WT transformation must include a WT column number where it needs to be applied and transformation type. We formalized these transformations as a higher-order-logic record called *trans_col_rec* with two fields named *trans* and *col* of type *transformation* and *nat* respectively. Data type *transformation* consists of two elements FA and HA. *trans* field represents the transformation type and *col* field represents the WT column number.

V. Wallace Tree Multiplier Verification

In this section, we present the verification of the correctness-preserving characteristics of the FA and HA transformations and the correctness of a WT synthesis process.

A. Correctness of the initial WT Structure

We first prove the correctness of our initial WT structure by proving its integer value to be equal to the product of the integer value of its operands

```
L1:∀ as bs i.eval_wts (wallace_tree as bs) i =
     (2^i) * (bv_to_nat as) * (bv_to_nat bs)
```

where, the function *bv_to_nat* converts a bit list to integer.

B. FA and HA transformations are correctness-preserving

The most vital step in our formal synthesis methodology is to establish the fact that all the synthesis transformations preserve the correctness of the initial model. A correctness-preserving synthesis transformation for a WT structure can be defined as the one that preserves the semantics of the initial WT structure. We proved that in the Isabelle/HOL theorem prover in two steps. In the first step, it is proved that both the FA and HA transformations preserve the integer value of any two WT structure columns.

```
L2_fa:∀ as bs i. eval_wts (fa_trans as bs) i =
       eval_wts (as#bs#[]) i
L2_ha:∀ as bs i. eval_wts (ha_trans as bs) i =
       eval_wts (as#bs#[]) i
```

Functions *fa_trans* and *ha_trans* accept two WT structure columns *as* and *bs* and replace the first 2 or 3 bits respectively of the first column *as* with their sum bit and append their carry bit to the top of the second column *bs*. Both functions return a partially compressed WT structure by concatenating the two modified columns. On the right hand side of the equations above, the two WT structure columns *as* and *bs* are simply concatenated. Function *eval_wts* is used on both sides to find the respective integer values.

In the second step, lemmas L2_fa and L2_ha are used to prove that both the FA and HA transformations preserve the integer value of the whole WT structure irrespective of its number of columns.

```
L3_fa:∀ as n i.eval_wts(fa_trans_tree as n) i =
       eval_wts as i
L3_ha:∀ as n i.eval_wts(ha_trans_tree as n) i =
       eval_wts as i
```

where functions *fa_trans_tree* and *ha_trans_tree* accept a WT structure *as* and a column number *n* and apply the functions *fa_trans* and *ha_trans* respectively on the n^{th} and $(n+1)^{th}$ columns in the WT structure *as*. Both these functions return the corresponding partially compressed WT structure. Function *eval_wts* is used to find the integer value of both the modified and the unmodified WT structures. Lemmas L3_fa and L3_ha prove that FA and HA transformations do not change the integer value of a WT structure and are thus correctness preserving.

C. Correctness of WT synthesis process

The WT synthesis process consists of applying a sequence of FA and HA transformations on the WT structure. This is modeled by a recursive function *apply_trans* that accepts a WT structure and a list of *trans_col_rec* records and applies all the transformations in the *trans_col_rec* list recursively. The final output of the *apply_trans* function is the final synthesized WT structure. The correctness theorem for the WT synthesis process can now be stated as follows:

```
T1: ∀ as bs ts i. eval_wts
(apply_trans(wallace_tree as bs) ts) i =
       (2^i) * (bv_to_nat as) * (bv_to_nat bs)
```

where *as* and *bs* are the multiplier operands and *ts* is the transformation sequence which is declared as a *trans_col_rec* list. The left hand side expression represents the integer value of the whole WT synthesis process, whereas the right hand side represents an integer multiplier. Thus, T1 can be used to guarantee correctness of any WT synthesis process independent of its operand widths or TT.

D. Successful Termination of WT synthesis process

The WT synthesis process is said to be successfully terminated if the post-synthesis height of the WT structure is reduced to 2, i.e., no column in the WT structure has a length more than 2. We developed a function *eval_fin_wal_tree* that checks for successful termination of a WT process by adding the integer value of the bit strings of the first two rows of the WT structure and checking it against the multiplication of the initial operands. The successful termination theorem can be stated as follows:

```
T2: ∀ as bs ts. (check_len_eq_2
 (apply_trans (wallace_tree as bs) ts)) ⇒
 eval_fin_wal_tree
   (apply_trans (wallace_tree as bs) ts) =
       (bv_to_nat as) * (bv_to_nat bs)
```

Theorem T2 cannot be proved in general as it is not valid for an unsuccessfully terminated WT synthesis process where the height of the final WT structure is more than 2. Therefore we proved it under the assumption of a successfully terminated WT synthesis process and the function *check_len_eq_2* returns *True* if and only if the length of any column in its WT structure argument is not more than two. The antecedent of the implication checks for successful termination and the conclusion of the implication states the correctness of the WT synthesis process. Theorem T2 represents all the conditions of a correctly synthesized WT multiplier and can be used to verify the correctness and termination of any WT synthesis process independent of its operand widths or TT.

VI. Synthesis of an MxN Multiplier

This section presents the working of our automated WT synthesis tool with the help of an example of synthesizing an MxN multiplier. The end user initiates the synthesis process by providing the operand

widths (M,N) to the synthesis tool. All the subsequent steps that are required to generate the gate-level netlist of the WT multiplier and the mathematical proof of its correctness are automated. The synthesis tool shown in Fig. 1 applies a compression algorithm to obtain the TT for the synthesis of a MxN WT multiplier. The TT is basically a sequence of HA and FA transformations that are required to reduce the height of the WT structure to two. This TT is used by the HDL translator and the higher-order-logic theory translator blocks shown in Fig. 1 to generate the HDL gate level netlist and the Isabelle/HOL lemma for the MxN multiplier correctness, respectively. The format of the generated correctness lemma and its proof steps is given below:

```
 lemma "eval_fin_wal_tree (apply_trans
(wallace_tree [(X_{m-1}:: bit), ... (X_0 :: bit)]
[(Y_{n-1}:: bit), ... (Y_0 :: bit)])
([(|trans = <HA or FA>, col=number|),... ]))=
bv_to_nat[X_{m-1},...X_0] * bv_to_nat[Y_{n-1},...Y_0]";
    apply (rule T2);
    apply (simp add: wallace_tree_def);
    apply (simp add: trans_tree_def);
```

The lemma explicitly states the operand widths and the sequence of transformations generated by the WT_compressor. The transformation sequence has been expressed as a list of *trans_col_rec* records. The lemma is followed by three proof steps. These steps have been designed in such a way that they can prove any lemma of this kind irrespective of the operand widths or the TT. This, in fact, leads to the automated formal synthesis without user intervention.

The Isabelle/HOL proof assistant loads the lemma and applies the three proof steps to prove it. The first proof step is to use theorem T2 as a simplification rule. T2 proves the correctness and successful termination of a WT synthesis process and is described in the previous section. Isabelle/HOL proof assistant uses the modus ponens inference rule along with T2 on the lemma and returns a sub goal that checks if the post synthesis height of the WT structure is two. Proof steps 2 and 3 add function definitions for *wallace_tree* and *trans_tree* respectively to the simplification rules. The Isabelle/HOL proof assistant obtains the initial WT structure for the given operand widths using the *wallace_tree* definition and applies the given sequence of transformations on this structure using the *trans_tree* definition. Thus the whole synthesis process is performed within the Isabelle/HOL core. The subgoal is successfully proved if, after all the transformations have been applied, the final WT structure does not contain a single column with a length greater than 2.

VII. Related Work

Formal synthesis is a promising approach and has been shown to work successfully at both the system [1] and algorithmic [3] levels. Several formal synthesis systems have been introduced such as, T-Ruby [12] and HASH [2]. Kumar et al. [9] present an interesting summary and classification of formal synthesis research activities. The proposed approach is primarily based on the formal synthesis concepts but also allows the end user to get the synthesis results along with their proof of correctness in an automated manner, as has been seen in this paper for the case of WT multiplier synthesis.

The post-synthesis verification of wide integer multipliers remained an open problem for a long time due to the state space explosion problem of the state based verification approaches [4]. Theorem proving based post-synthesis verification is capable of handling the computational complexity of wide multipliers [7] but such efforts have also been limited as they involve considerable end user intervention. A number of dedicated multiplier verification algorithms have been published in recent years which allow us to verify multipliers of arbitrary sizes [8]. Similarly, combinations of model checking and theorem proving techniques have also been successfully tried in this domain [6]. These recent approaches are even though capable of verifying wide multipliers but their complexity definitely increases with the increase in operand widths. On the other hand, the WT multiplier synthesis approach, presented in this paper, is capable of handling arbitrary operand widths without any change in the computation complexity levels.

VIII. Conclusions

We presented a formal synthesis methodology that can be automated and thus it not only ensures formally verified synthesis results but also is very easy to use for end users who do not have any background in formal semantics and reasoning. Our synthesis methodology achieves correctness by construction and thus eliminates the post synthesis verification requirements, which in turn reduces design time. We have demonstrated the practical effectiveness of our methodology by successfully constructing an automated tool that is capable of correctly synthesizing WT multipliers of arbitrary length operands. The proposed formal synthesis methodology is quite general and can be applied to correctly synthesize any digital circuit. As a future work for this project, it would be interesting to verify the correctness-preserving characteristic of synthesis transformations for other digital circuits as well. This way we will be able to enhance the library of formally verified correctness-preserving synthesis transformations and thus formally synthesize a bigger set of combinational digital circuits.

References

[1] C. Blumenröhr. A formal approach to specify and synthesize at the system level. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 1999.

[2] C. Blumenröhr, D. Eisenbiegler, and D. Schmid. On the efficiency of formal synthesis – experimental results. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems*, 18(1):25–32, 1999.

[3] C. Blumenröhr and V. Sabelfeld. Formal synthesis at the algorithmic level. In *CHARME*, 1999.

[4] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[5] A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. *HDL Descriptions to Gauranteed CorrectCircuit Designs*, pages 43–67, 1987.

[6] R. Kaivola and N. Narasimhan. Formal verification of the pentium 4 floating-point multiplier. In *DATE*, 2002.

[7] D. Kapur and M. Subramaniam. Mechanically verifying a family of multiplier circuits. In *LNCS 1102, Computer Aided Verification*, 1996.

[8] M. Keim, R. Drechsler, B. Becker, M. Martin, and P. Molitor. Polynomial formal verification of multipliers. *Formal Methods in System Design*, 22(1):39–58, 2003.

[9] R. Kumar, C. Blumenröhr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design-A classification and survey. In *First international conference on formal methods in computer-aided design*, volume 1166, pages 294–299, 1996.

[10] N. Narasimhan, E. Teicad, R. Radhakrishnan, S. Govindarajan, and R. Vemuri. Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. *Formal Methods in System Design*, 19(3):237–273, 2001.

[11] L. C. Paulson. Isabelle: A generic theroem prover. In *LNCS*, 1994.

[12] R. Sharp and O. Rasmussen. The T-Ruby design system. *Formal Methods in System Design: An International Journal*, 11(3):239–264, 1997.

[13] C . S. Wallace. A suggestion for a fast multiplier. *IEEE transactions in Electronic Computers*, 13, 1964.