PAPER *Special Section on Formal Approach*

A Unified Framework for Equivalence Verification of Datapath Oriented Applications

Bijan ALIZADEH[†a)], ***Member and*** **Masahiro FUJITA**[†], ***Nonmember***

SUMMARY In this paper, we introduce a unified framework based on a canonical decision diagram called Horner Expansion Diagram (HED) [1] for the purpose of equivalence checking of datapath oriented hardware designs in various design stages from an algorithmic description to the gate-level implementation. The HED is not only able to represent and manipulate algorithmic specifications in terms of polynomial expressions with modulo equivalence but also express bit level adder (BLA) description of gate-level implementations. Our HED can support modular arithmetic operations over integer rings of the form Z_{2^n}. The proposed techniques have successfully been applied to equivalence checking on industrial benchmarks. The experimental results on different applications have shown the significant advantages over existing bit-level and also word-level equivalence checking techniques.

key words: *equivalence verification, canonical form, RTL model, gate-level implementation, decision diagram*

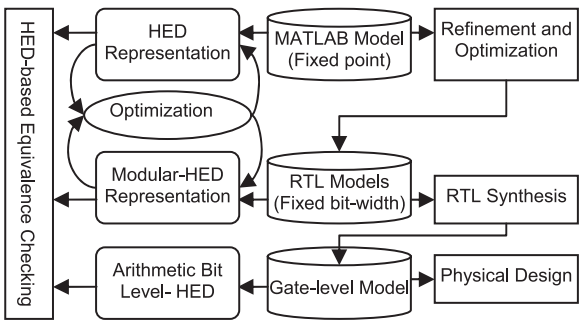

Fig. 1 HED-based equivalence checking with a complete design flow for different applications.

1. Introduction

As system on a chip (SoC) designs continue to increase in size and complexity, more attention has been paid to design descriptions at higher levels of abstraction due to faster design changes and higher simulation speed. In such cases, a C-based high level (an algorithmic level) specification is described and then refined to a Register Transfer Level (RTL) description by adding more and more implementation details at different steps. These refinement/optimization steps are performed manually or by using various automated tools such as [2]. Subsequently, gate-level implementations are synthesized using RTL synthesis tools. Therefore there is a significant increase in the amount of verification efforts to achieve functionally correct description at each step, if traditional dynamic techniques such as simulation are used. This has led to a trend away from dynamic approaches and therefore formal equivalence checking methods have become very important to reduce time-to-market as much as possible. Although contemporary verification approaches have proposed different modeling and manipulation methods for each stage of this complete design flow [3]–[5], there is no total and uniform solution to perform equivalence checking of different models in a unified representation.

Figure 1 depicts a complete design flow where algorithmic specifications usually implement sequences of arithmetic polynomial computations with integer variables of infinite bit-widths, which can be directly represented by HED [1]. After refining algorithmic specifications to RTL descriptions, it is evident that RTL models are implemented with fixed-bit-width datapath architectures, so that polynomial computations are carried out over *n-bit* integers where the sizes of the entire datapaths are kept constant by way of signal truncation. In this paper we provide an extension to the HED for supporting modular arithmetic polynomials which can directly manipulate these RTL design descriptions.

The final issue explored in this paper is the representation of gate level implementation. After synthesizing the RTL models into gate-level implementations, arithmetic bit level netlists extracted from the gate-level circuits are also represented in HED, which gives efficient ways for equivalence checking of designs in various design stages. We use an adder-extraction technique proposed in [3] for equivalence checking of arithmetic circuits based on a Bit-Level Adder (BLA) representation in order to efficiently represent gate-level circuits in HED. This technique benefits from a more efficient reverse engineering process compared to the conventional methods such as arithmetic bit-level (ABL) description in [4]. The BLA directly maps a gate-level description to a word-level addition network, while the ABL is an intermediate representation between gate-level and word-level descriptions and further reverse-engineering process is necessary to extract the word-level description from ABL representation. Moreover, the HED is strong enough to be used as a formal model for different levels of abstraction during refinements from high-level specification to RTL as well as gate-level implementation models.

The paper is organized as follows: Section 2 provides a brief review of related works. In Sect. 3 we briefly describe

Manuscript received July 24, 2008.
Manuscript revised October 13, 2008.

[†]The authors are with VLSI Design and Education Center (VDEC), The University of Tokyo and JST CREST, Tokyo, 113–0032 Japan.
a) E-mail: alizadeh@cad.t.u-tokyo.ac.jp

DOI: 10.1587/transinf.E92.D.985

the HED package as a canonical decision diagram. Section 4 presents how to implement modular polynomial as well as arithmetic bit level in HED. In Sect. 5 we discuss about experimental results, and finally concluding remarks and future works are shown in Sect. 6.

2. Related Work

In the literature of graph-based canonical representation, various extensions over the classical Binary Decision Diagram (BDD) introduced in [6] have been derived to reduce the size of the graph or to speed up the construction process. Although BDD and their variants have found wide application in formal verification methods, they suffer from memory explosion problems when the designs grow in size and complexity with arithmetic operations.

To alleviate this issue, *Word Level Decision Diagrams* (WLDDs) have been proposed. WLDDs are graph-based representations for functions with a Boolean domain and an integer range [5]. Furthermore, Binary Moment Diagrams (BMD), Multiplicative BMD (*BMD) and Kronecker *BMD (K*BMD) provide representations of integer-valued functions defined over bit-vectors and attempt to make the decomposition more efficient in terms of the graph size [7]. However, they fail to represent Boolean functions that can easily be represented using BDD. They still suffer from memory explosion when dealing with wide arithmetic operations due to defining functions over binary variables as a bit vector rather than integer variables.

In these approaches, BDD or WLDDs are utilized to represent symbolic expressions. However, system-level specifications such as those for digital signal processing contain a lot of arithmetic operations that must be encoded into bit level operations. Thus these techniques are not able to handle these designs due to the large number of Boolean variables. The latest word-level representation in the literature, i.e., Taylor Expansion Diagram (TED), supports multiplication and is able to represent functions with an integer domain and range [8]. It uses the Taylor series expansion as its decomposition method to represent a multivariate polynomial expression. However, TED does not model modulo arithmetic and therefore is not able to prove computational equivalence of polynomials over finite integer rings.

Verification approaches for bit-vector arithmetic such as term-rewriting, arithmetic decision procedures, polynomial decision diagrams and word-level ATPG have been studied in [9]–[11]. The authors in [11] have used integer arithmetic in constraint satisfaction for ILP-based simulation vector generation. However, these methods have tried to solve linear congruence using modulo arithmetic concepts and therefore are not applicable to prove polynomial equivalence modulo 2^n.

In [12], [13], the properties of polynomials over finite integer ring have been used to analyze and verify polynomial expressions with module 2^n. These are useful techniques for word-level reasoning. Their implementations are, however, based on manipulations of symbolic expressions and they are not applicable to bit level equivalence checking problems.

In [4], people have proposed a normalization technique for verifying arithmetic circuits in a bounded model checking environment. Their technique operates on the arithmetic bit level (ABL) description of a given circuit which contains three objects: partial product generator, addition network and comparator. These objects can always be decomposed into a netlist of half-adders. After extracting an ABL representation from the gate netlist, they generate the reduced normal form by applying a complex process to keep the intermediate size of the ABL description as small as possible. In this paper, however, we utilize a new technique presented in [3] which gives a scalable implementation for large industrial benchmarks as will be discussed in Sect. 4.2.

3. Horner Expansion Diagram (HED)

The goal of this section is to introduce a graph-based representation called HED [1] for functions with a mixed Boolean and integer domain and an integer range to represent arithmetic operations at a high level of abstraction, while other proposed *Word Level Decision Diagrams* (WLDDs) [5], [7] are graph-based representations for functions with a Boolean domain and an integer range. In HED, functions to be represented are maintained as a single graph in canonical form. We assume that the set of variables is totally ordered and that all of the vertices constructed obey this ordering. Maintaining a canonical form requires obeying a set of conventions for vertex creation as well as weight manipulation.

HED is a binary graph-based representation which supports polynomial function by factorizing variables recursively as shown in Eq. (1), where *const* is a term which is independent of variable X, while *linear* is another term which is served as the coefficient of variable X.

$$\begin{aligned} F(X,\ldots) &= F(X=0,\ldots) + X * [F'(X=0,\ldots) + \cdots] \\ &= const + X * linear \end{aligned} \tag{1}$$

In order to normalize the weights, any common factor is extracted by taking the greatest common divisor (gcd) of the argument weights. Once the weights have been normalized the hash table is searched for an existing vertex or creates a new one. Similar to that of BDDs, each entry in the hash table is indexed by a key formed from the variable and the two children, i.e. *const* and *linear* parts. As long as all vertices are created, the graph will remain in canonical form.

Example. Figure 2 illustrates how $f(X,Y,Z) = 24 - 8*Z + 12*Y*Z - 6*X^2 - 6*X^2*Z$ is represented by HED. Let the ordering of variables be X, Y, Z. First the decomposition w.r.t. variable X is taken into account. As shown in Fig. 2 (a), after rewriting $f(X,Y,Z) = (24 - 8Z + 12YZ) + X*(-6X - 6XZ)$ based on Eq. (1), *const* and *linear* parts will be $24-8*Z+12*Y*Z$ and $-6*X-6*X*Z$ respectively. The *linear* part is decomposed w.r.t. variable X again due to X^2 term. After that, the decomposition is performed w.r.t. variable Y and then Z

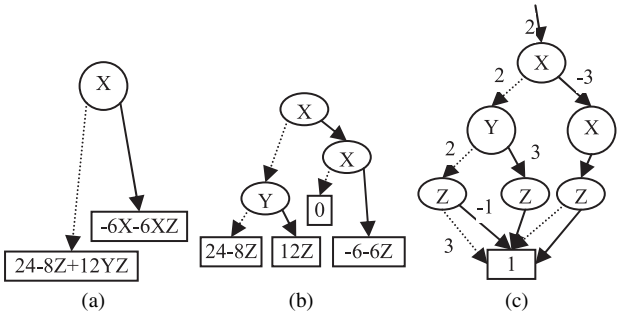

Fig. 2 HED representation of $24 - 8Z + 12YZ - 6X^2 - 6X^2Z$.

as shown in Fig. 2 (b). In order to reduce the size of an HED, redundant nodes are removed and isomorphic sub-graphs are merged. For this purpose the greatest common divisor of the argument weights are taken to figure out isomorphic sub-graphs as well as redundant nodes. In Fig. 2 (b), $24-8Z$, $12Z$ and $-6-6Z$ are rewritten by $8[3+Z*(-1)]$, $12[0+Z*(1)]$ and $-6[1 + Z * (1)]$ respectively. In order to normalize the weights, $\gcd(8, 12) = 4$ and $\gcd(0, -6) = -6$ are taken to extract common factors. Finally, Fig. 2 (c) shows the normalized graph where $\gcd(4, -6) = 2$ is taken to extract common factor between out-going edges from *X node*. In this representation, dashed and solid lines indicate *const* and *linear* parts respectively. Note that in order to have a simpler graph; paths to 0-terminal have not been drawn in Fig. 2 (c).

4. Extensions to HED

4.1 Modular Arithmetic in HED

In this section, we present how to implement modular multi-variate polynomials in HED by using ideas presented in [12], [13]. In order to keep the formulas short, we use the following multi-index notation. For $\mathbf{k} = \langle k_1, k_2, \ldots, k_d \rangle$ as the degrees corresponding to the d variables $\mathbf{x} := \langle x_1, x_2, \ldots, x_d \rangle$, let

$$\mathbf{x^k} := \prod_{i=1}^{d} x_i^{k_i} \quad \mathbf{k!} := \prod_{i=1}^{d} k_i! \quad \binom{x}{k} := \prod_{i=1}^{d} \binom{x_i}{k_i}$$

The *Smarandache function* $S(m)$ in number theory is defined for a given positive integer m as the smallest positive integer such that its factorial $S(m)!$ is dividable by m. For example, the number 8 does not divide 1!, 2!, 3!, but does divide 4!, so that $S(8) = 4$. Generally, in the ring of interest, Z_{2^n}, let $S(2^n) = k$, such that k is the smallest number satisfying $2^n \mid k!$. For example, $S(8 = 2^3) = 4$ as 8 divides $4! = 4 * 3 * 2 * 1 = 2^3 * 3$. This property can be used to find out shrinking polynomials from the original one.

Theorem 1: *The polynomial* $g(x) = \prod_{i=1}^{S(m)}(x + i)$ *is equivalent to* 0 *in* Z_m *and it is called vanishing polynomial. Here* $S(m)$ *denotes the Smarandache function.* The proof is available in [14].

This theory indicates that if we can factorize a polynomial function $g(x)$ into a product of $S(m)$ consecutive numbers, then $g(x)$ can be reduced to 0 in Z_m. For example, consider polynomial $f(x) = x^6 + 21x^5 + 175x^4 + 735x^3 + 1624x^2 + 1764x + 720$ over Z_{2^4}, where $S(2^4) = 6$. Since $f(x)$ can be described as a product of 6 consecutive numbers, i.e. $(x+6)(x+5)(x+4)(x+3)(x+2)(x+1)$, then $f(x) \bmod 2^4 \equiv 0$.

Lemma 1: *If* $2^n \mid a\mathbf{k!}$, *then* $a\mathbf{x^k}$ *is reducible modulo* 2^n. *The reduced polynomial is* $a\mathbf{x^k} - a * \prod_{j=1}^{d}\prod_{i=1}^{k_j}(x_j + i)$, *where* d *is the number of variables and* k_j *denotes the degree of* j^{th} *variable in the monomial* $\mathbf{x^k}$.

For example consider the monomial $f(x_1, x_2, x_3) = 2{x_1}^3{x_2}^2x_3$ in ${Z_2}^3$. Here $a = 2$ and $\mathbf{k!} = k_1!.k_2!.k_3! = 3!.2!.1! = 12$. Since $2^3 \mid 2 * 12$, therefore this monomial is reducible. On the other hand, based on a generalization of theorem 1 to multi-variate polynomials, we know that $V(x_1, x_2, x_3) = \prod_{j=1}^{3}\prod_{i=1}^{k_j}(x_j + i)$ is a vanishing polynomial in ${Z_2}^3$. Therefore, we can rewrite $f_{\text{REDUCED}}(x_1, x_2, x_3) = f(x_1, x_2, x_3) - a * V(x_1, x_2, x_3) = 2{x_1}^3{x_2}^2x_3 - 2(x_1)(x_1 + 1)(x_1 + 2)(x_2)(x_2+1)(x_3)$ to obtain a reduced polynomial function where the monomial $2{x_1}^3{x_2}^2x_3$ has been removed.

If $a\mathbf{k!}$ is not dividable by 2^n, although it is not possible to reduce the degrees of monomials, it might be possible to reduce the coefficient of the term of maximal degree. The following theorem helps us to do so, while the proof of this theorem is provided in [13].

Theorem 2: *Every polynomial* $f \in Z_{2^n}^d$ *has a unique representation of the form* $f(\mathbf{x}) = \sum_{\mathbf{k}\in N^d}\alpha_{\mathbf{k}}.\mathbf{x^k}$, *where* $\alpha_{\mathbf{k}} \in \{0, 1, \ldots, 2^{n-v2(\mathbf{k!})} - 1\}$ *and* $v2(\mathbf{k!}) < n$. $v2(\mathbf{k!})$ *is defined as the maximum degree* x *such that* 2^x *divides* $\mathbf{k!}$. *In other words,* $v2(\mathbf{k!})$ *gives the number of factors* 2 *in* $\mathbf{k!}$.

Now, let us consider those monomials in the function where $a\mathbf{k!}$ is not dividable by 2^n. In this case, if $a > 2^{n-v2(\mathbf{k!})} - 1$, it means that $a \notin \alpha_{\mathbf{k}}$ (from theorem 2) and therefore the coefficient a is reducible. We write the coefficient $a = q.2^{n-v2(\mathbf{k!})} + r$, where q is the quotient and r is the remainder. Therefore $a\mathbf{x^k} = q.2^{n-v2(\mathbf{k!})}.\mathbf{x^k} + r.\mathbf{x^k}$. It should be noted that the term $q.2^{n-v2(\mathbf{k!})}.\mathbf{x^k}$ is again reducible from lemma 1. The second term, i.e. $r.\mathbf{x^k}$, is already in reduced form since $r < 2^{n-v2(\mathbf{k!})}$. For example, consider the monomial $3{x_1}^3{x_2}^2x_3$ in ${Z_2}^3$, where $a = 3$ and $v2(\mathbf{k!}) = 2$ ($3!\,2!\,1!$ is dividable by 2^2). Since 2^3 does not divide $3.(3!\,2!)$, and $3 > 2^{3-2} - 1$ ($a > 2^{n-v2(\mathbf{k!})} - 1$), we represent $3{x_1}^3{x_2}^2x_3 = q.2^{3-2}.{x_1}^3{x_2}^2x_3 + r.{x_1}^3{x_2}^2x_3$, where $q = 1$ and $r = 1$. The monomial $2.{x_1}^3{x_2}^2x_3$ can be reduced to a lower total degree, as shown before, but ${x_1}^3{x_2}^2x_3$ is already in reduced form and further reduction is not possible.

Figure 3 depicts the algorithm for reducing a given polynomial into a unique form based on HED manipulations. If 2^n divides $a \times \overline{k}!$, the monomial is reduced as shown in line 11. Otherwise, if $a > \alpha_{\overline{k}}$, any monomial $a\overline{x}^{\overline{k}}$ is written as $q\times(\alpha_{\overline{k}}+1)\overline{x}^{\overline{k}} + r\times\overline{x}^{\overline{k}}$, where $q\times(\alpha_{\overline{k}}+1)\overline{x}^{\overline{k}}$ is reducible from *Lemma* 1 (line 14), while $r \times \overline{x}^{\overline{k}}$ is not reducible any more since $r < \alpha_{\overline{k}}$. Hence we consider the later term as a part of final result in another HED (*result* in line 15). If $a \le \alpha_{\overline{k}}$, we say the monomial is neither degree-reducible nor coefficient-reducible and the monomial is added to the

Modular_HED (HED poly, int n)
1 *poly* = multi-variate polynomial in HED before reduction;
2 *n* = the number of bits (variable's bit-width); result=0;
3 **for** each monomial *mon* (with the largest degree) in *poly* **do**
4 a = Coefficient (*mon*);
5 **for** each variable x_i **do**
6 k_i = Degree(x_i) in *mon;*
7 **end for**
8 $\bar{k}! = \prod k_i!; \quad v2(\bar{k}!) = \sum v2(k_i!)$;
9 $\alpha_{\bar{k}} = 2^{n-v2(\bar{k}!)} - 1$;
10 **if** ($2^n \mid a \times \bar{k}!$) **then** //*mon* is degree-reducible
11 $reducedQ = mon - a \times \prod_{j=1}^{d} \prod_{i=0}^{k_j-1} (x_j + i)$
12 **else if** ($a > \alpha_{\bar{k}}$) **then** //*mon* is coefficient-reducible
13 $q = a/(\alpha_{\bar{k}} + 1); r = a\%(\alpha_{\bar{k}} + 1)$;
14 $reducedQ = q(\alpha_{\bar{k}} + 1)(\prod_{j=1}^{d} x_j^{k_j} - \prod_{j=1}^{d} \prod_{i=0}^{k_j-1} (x_j + i))$
15 $result = result + r \times \prod_{j=1}^{d} x_j^{k_j}$;
16 **else** $reducedQ = 0$; $result = result + mon$;
17 **end if**
18 *updated_poly = poly – mon + reducedQ;*
19 **if** (*updated_poly* == 0) **then** **return** *result*;
20 **else** *poly = updated_poly*; **end if** **end for**
21 **return** *result;*

Fig. 3 Modular arithmetic in HED.

result (line 16). Finally the polynomial is updated with the reduced *mon* (line 18). This concept of monomial reduction is iteratively applied to a given polynomial, in order to reduce it to a unique form in HED according to *Theorem 2*. In the worst case, in each iteration, the *mon* computation requires $O(d * m)$ multiplications, where m is the maximum degree of each variable and d is the number of variables. Therefore, the worst case complexity of the algorithm is $O(d * m^{d+1})$.

Example: Let us consider the following polynomial which has 2 variables and 9 monomials and suppose variables' bit-width is 4 ($n = 4$):

$$poly = 4x^3y^3 + 17x^3y^2 + 12x^3y + 14x^2y^3 + 48x^2y^2 + 36x^2y + 13xy^3 + 32xy^2 + 24xy$$

Iteration 1. $mon = 4x^3y^3$ is taken into account since its variables have the largest degree w.r.t. those of other monomials and $a = 4$; $k_1 = 3$; $k_2 = 3$; $\mathbf{k!} = k_1! * k_2! = 3! * 3! = 36$. Since $a * \mathbf{k!}/2^n = 4 * 36/2^4 = 9$, the monomial is degree-reducible which means this monomial can be removed. The reduced monomial is computed as follows (line 11 in Fig. 3):

$$\begin{aligned} reducedQ &= 4x^3y^3 - 4(x)(x+1)(x+2)(y)(y+1)(y+2) \\ &= -12x^3y^2 - 8x^3y - 12x^2y^3 - 36x^2y^2 \\ &\quad - 24x^2y - 8xy^3 - 24xy^2 - 16xy \\ updated\_poly &= poly - mon + reducedQ \\ &= 5x^3y^2 + 4x^3y + 2x^2y^3 + 12x^2y^2 + 12x^2y \\ &\quad + 5xy^3 + 8xy^2 + 8xy \end{aligned}$$

Iteration 2. $poly = updated\_poly$ and $mon = 5x^3y^2$ are taken into account, where $a = 5$; $k_1 = 3$; $k_2 = 2$; $\mathbf{k!} = k_1! * k_2! = 3! * 2! = 12$; $v2(\mathbf{k!}) = 2$; $\alpha_{\mathbf{k}} = 2^{n-v2(\mathbf{k!})} - 1 = 2^{4-2} - 1 = 3$. Since $a * \mathbf{k!}/2^n = 5 * 12/2^4 = 15/4$, the monomial is not degree-reducible, however it is coefficient reducible because $a > \alpha_{\mathbf{k}}$ ($5 > 3$). Therefore we rewrite $5x^3y^2 = q * (\alpha_{\mathbf{k}} + 1) * x^3y^2 + r * x^3y^2$, where $q = r = 1$. The second term $r * x^3y^2 = x^3y^2$ is not reducible, while the first term $q * (\alpha_{\mathbf{k}} + 1) * x^3y^2 = 4x^3y^2$ can be removed as follows (lines 13–15 in Fig. 3):

$$\begin{aligned} reducedQ &= 4[x^3y^2 - x(x+1)(x+2)(y)(y+1)] \\ &= -4x^3y - 12x^2y^2 - 12x^2y - 8xy^2 - 8xy \\ result &= result + r * x^3y^2 = 0 + x^3y^2 = x^3y^2 \\ updated\_poly &= poly - mon + reducedQ = 2x^2y^3 + 5xy^3 \end{aligned}$$

Iteration 3. $poly = updated\_poly$ and $mon = 2x^2y^3$ are taken into account, where $a = 2$; $k_1 = 2$; $k_2 = 3$; $\mathbf{k!} = k_1! * k_2! = 2! * 3! = 12$; $v2(\mathbf{k!}) = 2$; $\alpha_{\mathbf{k}} = 2^{4-2} - 1 = 3$. Since $a * \mathbf{k!}/2^n = 2 * 12/2^4 = 3/2$ and $a < \alpha_{\mathbf{k}}$ ($2 < 3$) the monomial is neither degree-reducible nor coefficient reducible. Therefore this monomial can not be reduced (line 16 in Fig. 3):

$$\begin{aligned} reducedQ &= 0 \\ result &= result + mon = x^3y^2 + 2x^2y^3 \\ updated\_poly &= poly - mon + reducedQ = 5xy^3 \end{aligned}$$

Iteration 4. $poly = updated\_poly$ and $mon = 5xy^3$ are taken into account, where $a = 5$; $k_1 = 1$; $k_2 = 3$; $\mathbf{k!} = k_1! * k_2! = 1! * 3! = 6$; $v2(\mathbf{k!}) = 1$; $\alpha_{\mathbf{k}} = 2^{4-1} - 1 = 7$. Since $a * \mathbf{k!}/2^n = 5 * 6/2^4 = 15/8$ and $a < \alpha_{\mathbf{k}}$ ($5 < 7$) the monomial is neither degree-reducible nor coefficient reducible. Therefore the final polynomial is as follows:

$$result = result + mon = x^3y^2 + 2x^2y^3 + 5xy^3$$

4.2 Bit-Level Adder Extraction

Arithmetic functions in digital circuits are almost always implemented using addition as the base function. For instance, multiplication is based on addition where two stages are required. In the first stage, the partial products are generated which are the inputs to the second stage that is a collection of addition circuits. The question is which pairs of vectors have been added with each other in the addition network. The proposed method consists of an initialization phase, which will be followed by a stepwise word-level adder-extraction process. The adder-extraction approach maps the logic-optimized gate net-list to a global Bit-Level Adder (BLA) representation.

4.2.1 Definitions

In this section, we are going to present a compatible representation to cover all possible addition processes for a multiplier without exhaustively checking all of them. For this

purpose, the following definitions are used.

Definition1 (ADD_SET): *ADD_SET* is the set of word-level results obtained from extracted adders. In the initialization, this set consists of initial partial product vectors. Then after extracting each word-level adder, this set will be updated by merging the two added partial product vectors.

Definition2 (LSB_POS(X)): The function: $LSB\_POS(X)$ gives the Least-Significant Bit (LSB) position of X, which is a member of *ADD_SET*. In the initialization phase of an n-bit multiplier, where the members of *ADD_SET* are partial product vectors (P_i), the *LSB_POS* for a P_i can easily be calculated. If the partial product vectors are initially placed in *ADD_SET* in an ascending order in terms of their LSB positions, then the LSB position of P_i will be equal to i:

$$LSB\_POS(P_i) = i \tag{2}$$

Definition3 (MSB_POS(X)): The same function can be defined to determine the position of the Most-Significant Bit (MSB) of a member in *ADD_SET*. In the initialization phase of an n-bit multiplier, if the partial product vectors are sorted in an ascending order in terms of their LSB positions, we have:

$$MSB\_POS(P_i) = i + n - 1 \tag{3}$$

Definition2 and *Definition3* determine the range of bits in a vector, which can be either "1" or "0". Calculating the *MSB_POS* and *LSB_POS* functions for all the *ADD_SET* members makes it possible to evaluate the ABL representation of all the possible adder-extractions from the *ADD_SET* members.

4.2.2 ADD_SET Initialization

For each multiplier, there are two possible *ADD_SETs* in the initialization. Exchanging the input vectors: A and B results in two different partial product vector initializations. Consider we have all the bit-level partial products of A and B as inputs: A_iB_j ($i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$). It means that for an n-bit multiplier, there are n^2 numbers of available bit-level partial products. The initial word-level partial products can be arranged in two ways leading to two separate initial *ADD_SETs*. The two possible initial *ADD_SETs* are listed below, where a vector P_i has been expressed by a $1 \times n$ matrix.

$$ADD\_SET\#1 = \left\{ \hat{P}_i \;\middle|\; \hat{P}_i = [A_iB_n, A_iB_{n-1}, \ldots, A_iB_1] \right\} \tag{4}$$

$$ADD\_SET\#2 = \left\{ P_i \;\middle|\; P_i = [A_nB_i, A_{n-1}B_i, \ldots, A_1B_i] \right\} \quad (i = 1, 2, \ldots, n) \tag{5}$$

Our algorithm starts with the above two possible initial *ADD_SETs*. Then it will search for word-level adders using an efficient bit-level representation of adders. As soon as the first adder is extracted, one of the above *ADD_SETs*, which does not match the extracted adder, will be eliminated and the algorithm carries on with the other *ADD_SET*.

4.2.3 Example

Consider $F = A \times B + 3 \times C = P_1 + P_2 + P_3 + P_4 + C + 2 \times C$, where A, B and C are 4-bit unsigned integer vectors and P_i is a partial product vector, i.e., $P_i = A_i * B_i$. The algorithm starts with a set of vectors for the addition network, called *ADD_SET*. In this example the initial *ADD_SET* is $\{P_1, P_2, P_3, P_4, C, 2C\}$. Then a first-level XOR search will be executed to extract all the XOR terms, while their input signals are from the *ADD_SET*. Each extracted XOR term may refer to a word-level adder with respect to its input signals. In this way the extracted XOR terms will be categorized into some groups, in which each group refers to a specific word-level adder. Generally we can represent each word-level addition in an *ADD_SET* by the Bit-Level Adder (BLA) schematic in Fig. 4. If X_1 and X_2 are unsigned integers, we have:

$$k = LSB\_POS(X_2) - LSB\_POS(X_1) \tag{6}$$

$$FA\_NUM = MSB\_POS(X_1) - LSB\_POS(X_2) \tag{7}$$

$$HA\_NUM = MSB\_POS(X_2) - MSB\_POS(X_1) \tag{8}$$

Where *HA_NUM* and *FA_NUM* are the number of half-adders and full-adders in the BLA representation of $X_1 + X_2$ and $LSB/MSB\_POS(X_i)$ gives the Least/Most-significant bit position of X_i. Also we have:

$$LSB\_POS(X_1 + X_2) = \min\{LSB\_POS(X_1), LSB\_POS(X_2)\} \tag{9}$$

$$MSB\_POS(X_1 + X_2) = \max\{MSB\_POS(X_1), MSB\_POS(X_2)\} + \varepsilon \tag{10}$$

If adding X_1 and X_2, results in a carry overflow, ε in (10) will be equal to "1", otherwise it would be zero.

After the first-level XOR extraction and categorizing them based on their references adder representation in Fig. 4, the number of XOR terms in each category must be equal to *FA_NUM* + 1. Therefore, if this condition is true,

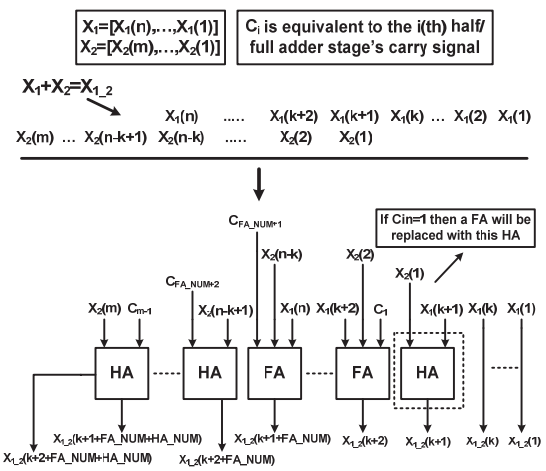

Fig. 4 BLA representation.

the equivalence checking of next-level XOR terms and carry signals will be executed to map the net-list to a dynamically built reference adder. If this process fails the whole net-list is not equivalent to the high-level description. This issue provides a fast convergence for the proposed method. Otherwise, if the XOR category does not include $FA\_NUM + 1$ number of XOR terms, the category will be rejected and they will be merged with the new extracted XOR terms in the next adder-extraction iterations to complete the new XOR categories. This verification process succeeds if the whole net-list is mapped to the high-level description.

As an example consider the addition network of $P_1 + P_2 + P_3 + P_4$ in Fig. 5, which refers to a 4-bit unsigned integer multiplier. In the first iteration five first-level XOR gates will be extracted. These XOR terms are labeled by $X_{i,j}$, which refers to the j^{th} first-level XOR gate extracted in the i^{th} iteration. The non-labeled XOR gates in the net-list are the next-level XORs. The other parameter $P_i(j)$ is the j^{th} MSB of the i^{th} partial product vector in *ADD_SET*. The XOR terms can be divided into three categories. The algorithm starts evaluating each category by assigning it a reference adder. The first category is $\{X_{1,1}\}$, which refers to the word-level addition of $P_1 + 2P_2$. The evaluation of this category easily results in its rejection as $FA\_NUM + 1 = 3 \neq 1$ based on (7) above.

Then the algorithm evaluates the second XOR category: $\{X_{1,2}, X_{1,3}, X_{1,4}\}$, which refers to $P_2 + 2P_3$ and satisfies

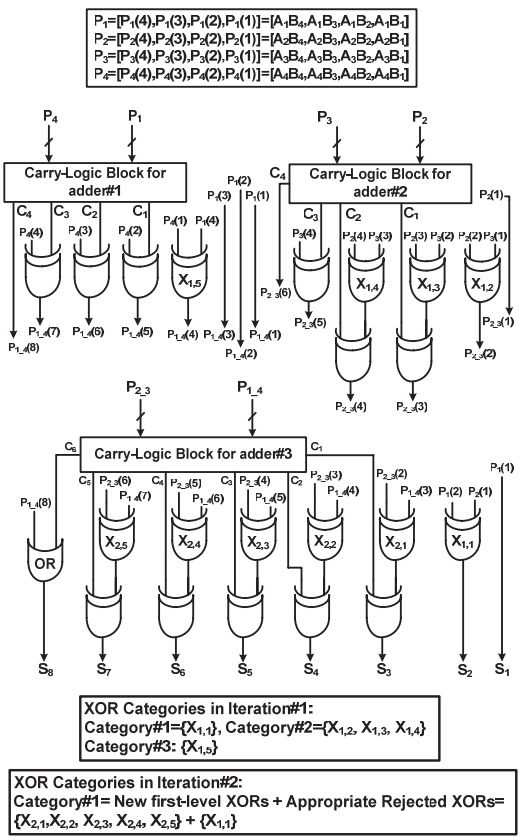

Fig. 5 Mapping the word-level adders to a 4-bit unsigned multiplier.

the condition, *i.e.* $FA\_NUM + 1 = 3$. Therefore, the next-level XORs will be extracted after performing the equivalence checking between each carry signal in the BLA model and the original circuit from LSB to MSB. After mapping the reference adder to the original circuit, *ADD_SET* will be updated to $\{P_1, P_{2\_3}, P_4\}$, in which we have:

$$P_{2\_3} = P_2 + P_3 = [P_{2\_3}(6), \ldots, P_{2\_3}(1)]$$
$$MSB\_POS(P_{2\_3}) = \max\{n+1, n+2\} + 1 = 7$$
$$LSB\_POS(P_{2\_3}) = \min\{2, 3\} = 2, \quad k(P_{2\_3}) = 3 - 2 = 1$$

The same process will be applied for the third category: $\{X_{1,5}\}$, and the second adder will be extracted. The second extracted adder calculates $P_1 + 8P_4$. As a result the next updated *ADD_SET* will be equal to $\{P_{1\_4}, P_{2\_3}\}$. After evaluating the first iteration's XOR terms, the second iteration begins and extracts five first-level XOR terms, which are also shown in Fig. 5. All these XOR terms are mapped to one category, which refers to $P_{1\_4} + 2P_{2\_3}$. It is interesting that the value of FA_NUM+1 for the equivalent adder of this category is 6 and the previously rejected XOR term: $X_{1,1}$, will complete the category. After mapping the carry signals, the final adder will be extracted from the net-list and the final *ADD_SET* equal to $\{P_{1\_2\_3\_4}\}$ will be obtained. Note that for an n-bit booth-encoded multiplier, *i.e,* $A \times B$, we have the following initial product vectors for the addition network:

$$ADD\_SET = \{P_m \,|\, P_m = (-2A_i + A_{i-1} + A_{i-2}) \times B\} \quad (11)$$
$$(i = 2m + 1 \leq n),\ (A_{-1} = 0,\ m = 0, 1, 2, \ldots)$$

With the formula above, we can extract and map adders in a similar way as shown above.

5. Experimental Results

In this section, we report two types of experiments that show the superiority of the *HED* compared to other approaches. In the first experiment, we have run *Modular HED* on some benchmarks in order to compare the results with those of proposed method in [12] as well as BDD, *BMD, SAT-miniSat [15] and MILP [11]. In another experiment, we have tried to represent BLA descriptions in HED. We implemented the HED package in C++ and carried out on an Intel 1.8 GHz Core 2 and 1 GByte of main memory running Windows XP.

5.1 Comparison with Other Technique

In this experiment, we follow up on comparing our approach with CUDD-BDD, *BMD, SAT-miniSat and MILP by employing phase-shift keying (PSK) used in digital communication, anti-aliasing functions (AAF), digital image rejection unit (DIRU), Degree-4 Filter (D4F) Savitzky Golay (SG) filter, MIBENCH (MI) polynomial used in automotive applications, Chebyshev (CHEB) polynomial and Quartic Spline (QS) benchmarks [12]. For each test-case, we are given two descriptions to be verified whether or not they are equivalent. Some of these designs were available as RTL

Table 1 Modular HED versus contemporary approaches.

Benchmark	Specs	Modular-HED	Technique [12]	CUDD-BDD	*BMD	miniSat [15]	MILP [11]
	Var/Deg/n	Nodes/Time (s)	Time (s)*	Nodes/Time (s)	Nodes/Time	Vars/Clauses/Tim	Time (s)
AAF	1 / 6 / 16	8 / 0.016	6.81	1.1M / 32.2	NA / >500	3.9K / 107K / >500	>500
D4F	1 / 4 / 16	6 / 0.031	4.95	27M / 20.3	NA / >1000	25K / 76K / >1000	>1000
CHEB	1 / 5 / 16	7 / 0.01	5.95	1M / 26.9	NA / >500	3.5K / 86K / >500	>500
PSK	2 / 4 / 16	16 / 0.032	13.48	NA / >500	NA / >500	52K / 142K / >500	>500
DIRU	2 / 4 / 16	9 / 0.016	14.4	NA / >1000	NA / >1000	10K / 30K / >1000	>1000
MI	2 / 9 / 16	26 / 0.2	17.5	23M / 39.4	NA / >1000	24K / 69K / >1000	>1000
SG	5 / 3 / 16	35 / 0.24	6.1	NA / >1000	NA / >1000	64K / 190K / >1000	>1000
QS	7 / 4 / 16	19 / 0.09	32.4	NA / >1000	NA / >1000	76K / 211K / >1000	>1000

NA: Not Applicable; K: Thousand; M: Million * from [12] and adjusted to CPU speed of 1.8 GHz

code, while the others were available as high-level description in C code. The related RTL codes for these high-level descriptions were automatically generated using high-level synthesis tools. After that, two descriptions are represented in HED and reduction procedure has been applied simultaneously in order to find out whether or not two descriptions are equivalent.

It was found that BDD could solve the problem for AAF, D4F, MI and CHEB benchmarks, but it failed for PSK, DIRU, SG and QS test-cases. For all benchmarks in Table 1, miniSat, *BMD and MILP could not solve the problem within the time-limit of 500 or 1000 seconds. The *BMD is a word-level decision diagram which is able to represent functions with Boolean domain and an integer range, while HED is capable to represent functions with hybrid Boolean/integer domains and integer ranges. Therefore, when the degree (k) of a polynomial increases, *BMDs are not satisfactory due to the fact that their size increases $O(n^k)$, where n is the bit-vector size. In our benchmarks the bit-vector size is 16. For instance consider $F = A * B$ where A and B are 16-bits wide. To construct *BMD, we have to take into account $A = 2^{15} * a_{15} + \cdots + a_0$ and $B = 2^{15} * b_{15} + \cdots + b_0$ and then compute $A * B$ w.r.t. bit-level variables (a_i and b_i for $i = 0$ to 15). It is clear that the number of nodes will be increased rapidly due to bit-level variables. However in HED, we just need to represent for example $A * B$, instead of $(2^{n-1} * a_{n-1} + \cdots + a_0) * (2^{n-1} * b_{n-1} + \cdots + b_0)$, where A and B are taken into account as word-level variables. Obviously, increasing the number of bits (bit-width) only increases the amount of computations needed to reduce polynomials as mentioned in Sect. 4.1.

Moreover, our approach is not only faster than BDDs, *BMDs, miniSat and MILP based equivalence checking techniques, but also reports better performance in comparison with the method in [12] which is based on symbolic algebra tools such as MAPLE. For instance, consider DIRU benchmark. In *Modular-HED*, the CPU time required to check the equivalence of the given designs is 0.016 s which is much less than 14.4 s of the proposed method in [12]. Obviously, the performance differences are multiple orders of magnitude due to the differences between bit-wise analysis and word-wise analysis. Also the proposed method in [12] needs to call MAPLE for each computation while in our approach all computations are carried out internally. In addition, after reducing the polynomials in *Modular-HED*, equivalent polynomials are automatically detected where the computation time is $O(1)$. If two polynomials F_1 and F_2 are not equivalent, $F_1 - F_2$ in HED, symbolically represents the difference between two functions which is directly related to the source of bugs. It may not be easy to find out bugs if symbolic algebra tools have been employed. In order to prove our claim we need to give more experimental results. However, we leave it as a future work to extend our approach for arithmetic circuit debugging.

5.2 Scalability of Our Approach

In the second experiment, we have generated various polynomials which differ from each other in terms of the maximum degrees and the number of variables. We are given a description according to Eq. (12), where d is the number of variables which varies from 2 to 8, while *Deg* indicates degree of different polynomials. Similar to the first experiment, for all benchmarks the bit-length n is 16. For example if $d = 2$ and $Deg = 4$ we will have $(x_1)(x_1+1)(x_1+2)(x_1+3)$ $(x_2)(x_2+1)(x_2+2)(x_2+3)$.

$$polynomial = \prod_{j=1}^{d} \prod_{i=0}^{Deg-1} (x_j + i) \tag{12}$$

After representing each polynomial in HED and then applying *Modular_HED* (Fig. 3), the experimental results are summarized in Table 2 in comparison with those of symbolic algebra tools like MAPLE. For most benchmarks in Table 2, the method presented in [12] could not solve the problem within the time-limit of 1200 seconds. In this table, the CPU time (*Time*) in seconds, memory usage (*Mem*) in MBytes and the number of HED nodes (*Nodes*) are reported for different polynomials, where d and *Deg* give the number of variables and degree of polynomials respectively. It should be noted that the number of nodes for symbolic algebra tools is not applicable (NA) and we could not report the memory usage. For instance, consider the last row in this

Table 2 Scalability of Modular-HED for different number of variables and degrees in comparison with symbolic algebra techniques.

Time (s) / Mem (MB) / Nodes					
Deg		**d = 2**	**d = 4**	**d = 6**	**d = 8**
8	Symbolic Algebra in [12]	5.7 / NA / NA	101 / NA / NA	579 / NA / NA	853 / NA / NA
	HED	0.5 / 0.7 / 72	8.1 / 6.2 / 732	35 / 20 / 2788	61 / 33 / 4992
12	Symbolic Algebra in [12]	12.4 / NA / NA	251 / NA / NA	791 / NA / NA	>1200 / NA / NA
	HED	1.7 / 1.9 / 156	21 / 13 / 1530	82 / 41 / 6890	126 / 59 / 10045
16	Symbolic Algebra in [12]	34.9 / NA / NA	644 / NA / NA	>1200 / NA / NA	>1200 / NA / NA
	HED	3.8 / 2.4 / 272	53 / 28 / 4468	184 / 77 / 13592	287 / 97 / 21974
20	Symbolic Algebra in [12]	70.7 / NA / NA	>1200 / NA / NA	>1200 / NA / NA	>1200 / NA / NA
	HED	7.3 / 4.9 / 420	138 / 59 / 10028	328 / 129 / 27012	467 / 157 / 48504

table, where *Time*, *Mem* and *Nodes* for a polynomial of degree 20 have been provided. In this case, $\prod_{j=1}^{d}\prod_{i=0}^{Deg-1}(x_j+i)$ is given where the number of variables (d) varies from 2 to 8, while *Deg* is 20. The results show that symbolic algebra approach [12] is only capable to solve the problem when the number of variables (d) is 2.

It is obvious that in our approach equivalent polynomials are automatically detected due to canonical representation of HED. However symbolic algebra tools such as MAPLE need to check the equivalence polynomials through some computations which spend lots of time and therefore will not be efficient. Furthermore, the HED package can be encapsulated into any design flow in order to check the equivalence between different levels of abstractions, specifically when we follow up a refinement-based design flow.

Table 3 Our approach versus ABL in [4].

Multiplier	**#of Evaluated Gates**		**Carry-Signal issues in [4]**		
	ABL [4]	Our Method	#HC	#NFA C	#ORMiss
8-bit-ripple	357	312	15	7	3
8-bit-CLA	577	335	15	7	3
16-bit-ripple	2037	1392	40	15	7
16-bit-CLA	2559	1736	40	15	7
32-bit-ripple	7747	5856	100	31	15
32-bit-CLA	10751	6802	100	31	15

5.3 HED for Gate-Level Implementation

In order to have a practical comparison between the proposed algorithm and the method in [4], we have synthesized the speed-optimized multipliers using the Xilinx ISE-9 synthesis tool on the XC95288XV CPLD device. The Xilinx tool applies a CLA-adder structure to speed-optimized multipliers. On the other hand the area- optimization process with ripple-carry adders has been applied to *c6288* from ISCAS-85 benchmark using the Synopsys Design Compiler tool. Table 3 tabulates the experimental results, where *m-bit-ripple* is the *m*-bit area-optimized multiplier, while the *m-bit-CLA* is the *m*-bit speed-optimized multiplier. The method in [4] performs the XOR search within all the internal nodes and therefore it has to search the XOR gates within the carry-signal logic blocks as well as equivalent XOR blocks. This process can become really time-consuming in large multipliers, due to the carry look-ahead logic block expansions. On the other hand the proposed method performs the first-level XOR search and does not proceed in the carry look-ahead logic. It means that the XOR search process in the proposed algorithm is almost similar for CLA and ripple-carry adders in terms of speed. The "*#evaluated gates*" column in Table 3 addresses the above issue.

The method in [4] evaluates all the logic gates in the net-list during XOR search and as a result "*#evaluated gates*" for this method is equal to total number of logic gates in the multiplier net-list. However, the proposed algorithm evaluates much lower number of gates during the XOR search, specifically, in large multipliers. Therefore, the XOR search process is much faster in the proposed method. Another factor, which makes the proposed algorithm much faster than the counterparts, is the process of categorizing XOR gates as explained in Sect. 4.2. By the time the algorithm fails to extract a word-level adder from an accepted XOR category, some bugs take place and therefore it stops the arithmetic verification process and reports the location of bugs. Therefore, the proposed algorithm converges really fast, even if it could not extract the equivalent arithmetic circuit from the gate net-list.

The equivalence checking of carry signals takes place for both methods. However, the method in [4] has to exhaustively check all the possible *non-fulladder* carries, *hidden carries* and full-adder carry signals for each extracted XOR gate. Our method can observe if two cascaded XOR terms should be taken into account as two half-adders or a single full-adder needs to be extracted. The other inefficiency of the ABL method in [4] was the unmapped OR gates. In the BLA, we represent each product vector by *LSB* and *MSB* functions and therefore each addition process can

Table 4 Multipliers in HED versus ABL [4].

#Bits		HED			ABL [4]	
		#NN	Time(s)	Mem(MB)	Time(s)	#Gate
8	WithoutEC	16	0.07	0.11	NA	NA
	With EC	29	0.37	0.3	1.92	577
16	WithoutEC	32	0.34	0.21	NA	NA
	With EC	61	1.9	0.98	6.56	2559
32	WithoutEC	64	1.38	0.7	NA	NA
	With EC	619	11.6	4.45	28.2	10751
64	WithoutEC	128	7.1	3.05	NA	NA
	With EC	2799	41.2	18.9	124.2	35936

NA: Not Applicable

be evaluated whether or not it leads to a carry overflow. In this way for each adder while evaluating the final half-adder stage in the net-list, we do know that if the OR gate, instead of XOR, is possible for the realization or not. Table 3 also addresses these problems by the three columns, the number of hidden-carry signals (*#HC*), the number of non-full-adder carry misses (*#NFAC*) and the number of unmapped OR gates (*#ORMiss*). As the multiplier becomes larger, these issues become more troublesome and the method in [4] has to exhaustively check different types of carry signals. The number of unmapped OR gates will also be increased in larger multipliers.

Furthermore, we have utilized the HED in order to verify the gate-level implementation of $n * n$ multiplier, where n varies from 8 to 64. The experimental results of equivalence checking between the gate-level description and a high level specification are summarized in Table 4. In this table, the number of bits, the number of HED nodes, CPU time and memory usage (during processing the gate level netlist) are reported in columns *#Bits*, *#NN*, *Time* (in second) and *Mem* (in MByte) respectively. Although the row *WithoutEC* tabulates those information just for representing high level specification A*B in HED, the row *With EC* reports them for equivalence verification of two descriptions. The last column in this table reports the CPU time required to run the ABL method which proves our claim that this method is time-consuming in large multipliers. In Table 4, column *#Gate* reports the number of gates to be processed in ABL method [4] which similar to *#of Evaluated Gates* column in Table 3 when m-bit CLA structures are considered. For instance, consider the last two rows in Table 4, where 64-bit multiplier was verified. The ABL requires 124.2 s (*With EC* row) to check the equivalence between the two descriptions. The HED package consumes 3.05 MB memory and spends 7.1 s run time to represent A*B high level description, where $A = \sum_{i=0}^{63} 2^i * a_i$; $B = \sum_{i=0}^{63} 2^i * b_i$. While it requires 18.9 MB memory and 41.2 s CPU time to check the equivalence between the bit level adder description extracted from the gate level net-list using our proposed method and a high level description, i.e., A*B. Obviously, the HED package spends some time to figure out isomorphic sub-graphs and redundant nodes which is $41.2 - 7.1 = 34.1$ s for 64-bit multiplier.

6. Conclusion and Future Work

In this paper we have proposed a unified framework to verify the equivalence between different levels of abstractions from a high level specification to a gate level implementation. We introduced the HED as a canonical decision diagram that not only supports modular polynomial computations over ring Z_{2^n}, but also is able to express ABL description of gate level implementations. We are interested in applying HED to the verification of arithmetic datapath computations over bit-vectors that have different bit-widths. Another future work is to diagnose and locate the source of bugs when the equivalence checking fails.

References

[1] B. Alizadeh and M. Fujita, "A canonical and compact hybrid word-Boolean representation as a formal model for hardware/software co-designs," Fourth Workshop on Constraints in Formal Verification (CFV), pp.15–29, 2007.
[2] I.A. Groute and K. Keane, "M(VH)DL: A MATLAB to VHDL conversion toolbox for digital control," IFAC Symp. on Computer-Aided Control System Design, Sept. 2000.
[3] O. Sarbishei, B. Alizadeh, and M. Fujita, "Arithmetic circuit verification without looking for internal equivalences," IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), June 2008.
[4] M. Wedler, D. Stoffel, R. Brinkmann, and W. Kunz, "A normalization method for arithmetic datapath verification," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.26, no.11, pp.1909–1922, Nov. 2007.
[5] S. Horeth and R. Drechsler, "Formal verification of word-level specifications," Proc. Design Automation and Test in Europe (DATE), pp.52–58, 1999.
[6] R. Bryant, "Graph-based algorithms for Boolean function manipulation," IEEE Trans. Comput., vol.35, no.8, pp.677–691, 1986.
[7] R. Drechsler, B. Becher, and S. Ruppertz, "The K*BMD: A verification data structure," IEEE Des. Test Comput., vol.14, pp.51–59, 1997.
[8] M. Ciesielski, P. Kalla, and S. Askar, "Taylor expansion diagrams: A canonical representation for verification of data flow designs," IEEE Trans. Comput., vol.55, no.9, pp.1188–1201, 2006.
[9] B. Alizadeh and M. Fujita, "Automatic merge-point detection for sequential equivalence checking of system-level and RTL descriptions," International Symposium on Automated Technology for Verification and Analysis (ATVA), LNCS 4762, pp.129–144, 2007.
[10] C.-Y. Huang and K.-T. Cheng, "Using word-level ATPG and modular arithmetic constraint solving techniques for assertion property checking," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.20, no.3, pp.381–391, 2001.
[11] R. Brinkmann and R. Drechsler, "RTL-datapath verification using integer linear programming," Proc. ASP-DAC, pp.741–747, 2002.
[12] N. Shekhar, P. Kalla, and F. Enescu, "Equivalence verification of polynomial datapaths using ideal membership testing," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.26, no.7, pp.1320–1330, 2007.
[13] N. Hungerbuhler and E. Specker, "A generalization of the smarandache function to several variables," Electronic Journal of Combinatorial Number Theory, vol.6, pp.A23, 1–11, 2006.
[14] L. Halbeisen, N. HungerBuhler, and H. Lauchli, "Powers and polynomials in Z_m," Elem. Math, vol.54, pp.118–129, 1999.
[15] http://minisat.se/

Bijan Alizadeh received his B.S., M.S. and PhD. degrees in computer engineering from the University of Tehran, Iran in 1995, 1998 and 2004, respectively. From 2004 to 2006 he was an adjunct Professor of Electrical Engineering Department at the Sharif University of Technology. In Fall 2006, he was a postdoctoral fellow at the VLSI Design and Education Center (VDEC) in the University of Tokyo in Japan. His research interests are in fundamental CAD techniques for verification, synthesis, optimization as well as test generation of digital VLSI circuits and systems.

Masahiro Fujita received his Ph.D. degree in Engineering from the University of Tokyo in 1985 and shortly after joined Fujitsu Laboratories Ltd. From 1993 to 2000 he had been assigned to Fujitsu's US research office and directed the CAD research group. In March 2000, he joined the department of Electronic Engineering in the University of Tokyo as a professor, and now a professor at VLSI Design & Education Center in the University of Tokyo. He has been involved in many research projects on various aspects of formal verification.