

LTED: A Canonical and Compact Hybrid Word-Boolean Representation as a Formal Model for Hardware/Software Co-designs

Bijan Alizadeh¹, and Masahiro Fujita¹

¹ VLSI Design and Education Center (VDEC), University of Tokyo, Japan
alizadeh@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract. In this paper, we describe a hybrid bit- and word-level representation called Linear Taylor Expansion Diagram (LTED) [1]. This representation not only has a compact and canonical form, but also is close to hardware-software descriptions so that it can be utilized as a formal model for formal verification and synthesis. The power of abstraction of LTED allows us to represent system level of hardware/software co-designs as well as implementation designs efficiently with run time requirements several orders of magnitude smaller than those of other known bit and word level representations discussed in the literature of formal verification.

In order to evaluate the performance of LTED as a hybrid representation, it is run on some industrial designs and experimental results are compared with Taylor Expansion Diagram (TED) which is also a word level canonical representation [2].

Keywords: Canonical Representation, Hybrid Solver, Hardware-Software Co-design, System-level Description, Register Transfer Level (RTL).

1 Introduction

A growing demand has emerged to design hardware at higher levels of abstraction due to rising technological complexity of system on a chip (SoC), high performance requirements and shortened time-to-market. Although many companies have adopted a methodology to start with a design specification using system-level languages like SystemVerilog, SpecC or SystemC, most of automated formal tools for high-level synthesis, property checking and equivalence checking are based on Boolean proof techniques such as BDDs and SAT solvers. Since arithmetic functions need to be encoded to Boolean expressions, these techniques suffer from memory and computational explosion problems respectively when dealing with wide arithmetic operations such as multiplications.

In order to cope with this limitation, a hybrid representation of *Word* and *Boolean* variables is expected to support word-level arithmetic functions as well as Boolean expressions. In the literature of canonical graph-based representation, we find a canonical word level representation called TED that is able to represent functions with an integer domain and range [2]. While other representations can only define integer-valued functions over binary variables as a bit vector, TED uses Taylor series as its decomposition method (for more details see [2]). In contrast to BDDs and

BMDs, TED is based on non-binary decomposition principle where the decomposition at each node leads to more than two terms. Although it is very suitable to represent polynomial expressions at algorithmic level descriptions, it requires longer run time and memory in comparison with our proposed representation (LTED) to represent implementation designs that contain logical operations as well as arithmetic operations.

The main contributions of our work in this paper are as follows:

- We propose a canonical representation of functions with a mixed Boolean and integer domain, and an integer range that is also compact enough due to utilizing a hybrid decomposition technique that will be discussed in Section 3.
- It allows to tightly combine bit- and word-level variables that make it strong enough to be used as a formal model for high-level and implementation descriptions. We call it Linear Taylor Expansion Diagram (LTED) because first-order linearization of the Taylor series expansion is exploited to represent algebraic expressions.
- LTED exploits nodes with only two edges rather than complex nodes with more than two edges used in TED which affect both the run time and memory usage. This is why the TED package spends more than 60 percent of the run time to find out redundant nodes and isomorphic sub-graphs.

The rest of this paper is organized as follows. Related works are addressed in Section 2. LTED as a hybrid canonical representation is described in Section 3. Experimental results of industrial case studies are discussed in Section 4. Finally a brief conclusion and future work are given in Section 5.

2 Related Works

In the literature of graph-based canonical representation, some extensions of the classical BDD introduced in [3] have been proposed to reduce the size of the graph [4] or to speed up the construction process [5]. Although BDDs 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.

To alleviate this issue, *Word Level Decision Diagrams* (WLDDs) have been proposed. The WLDDs are graph-based representations for functions with a Boolean domain and an integer range [6], [7]. Furthermore, Binary Moment Diagrams (BMDs), Multiplicative BMDs (*BMDs) and Kronecker *BMDs (K*BMDs) provided a representation of integer-valued functions defined over bit-vectors and attempted to make the decomposition more efficient in terms of the graph size [8]. Using *BMDs it was for the first time possible to verify multipliers of bit length up to $n=256$, however, they fail for the representation of Boolean functions that can easily be represented using BDDs. A thorough review of different *Decision Diagrams* can be found in [9]. 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.

Another methodology transforms the design into propositional logic formulas and then satisfiability tools, i.e., SAT or SMT solvers, are employed to verify the validity of the formulas [10], [11]. Although Boolean SAT-based methods are very efficient

regarding memory requirement, they have not been very successful for designs containing large arithmetic units due to computational complexity.

In [12], [13] a hybrid satisfiability approach (HSAT) has been introduced to generate functional test vectors for RTL designs. This approach creates linear arithmetic constraints for arithmetic operators and conjunctive normal form (CNF) clauses for Boolean logical operators. Then it uses 3-SAT checking to solve the logic equations and integer linear programming (ILP) solver to check the feasibility of the arithmetic equations separately, in different domains. Interactions between Boolean variables in the CNF formulas and integer variables in arithmetic equations are handled by increasing the number of integer variables while reducing their range. Hence for variables that correspond to the interaction between the Boolean and arithmetic domains of the design, an assignment is selected from the CNF-clauses, and the resulting constraints are propagated to the arithmetic domain to check for consistency. If variable assignments that satisfy the CNF clauses cause the linear programming constraints in the arithmetic domain to be infeasible, backtracking is needed to select another set of Boolean assignments. Since these two engines operate in separate domains, the performance of HSAT is limited by the heuristics that choose the set of assignments to Boolean variables. Moreover, although HSAT is able to model bit-level and word-level expressions, nonlinear operators such as integer multiplication should be decomposed into linear operators due to using integer linear programming. To do so, one of the operands needs to be expanded in terms of bit-level variables.

In all above approaches BDD, WLDD or SAT based methods are utilized to represent symbolic expressions while system-level specifications such as those for digital signal processing contain a lot of arithmetic operations that should be encoded into bit level operations. Thus these techniques are not able to handle these designs due to the large number of Boolean variables or clauses. Moreover, HSAT only deals with scalar multiplication while the last 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 [2]. It uses the Taylor series expansion as its decomposition method and represents a multivariate polynomial expression. Let $f(x,y,\dots)$ be a real differentiable function in variables $\{x, y, \dots\}$. Using the Taylor series expansion with respect to a variable x , function f can be represented as follows:

$$f(x, y, \dots) = f(x=0, y, \dots) + xf'(x=0, y, \dots) + \frac{1}{2}x^2 f''(x=0, y, \dots) + \dots$$

The derivation of f evaluated at $x=0$ are independent of variable x and can be further decomposed with respect to the remaining variables, one variable at a time. The resulting recursive decomposition can be represented by a decomposition diagram, called the *Taylor Expansion Diagram* or TED. In contrast to BDDs and BMDs, TED is based on non-binary decomposition principle where the decomposition at each node leads to more than two terms. Although it is very suitable to represent polynomial expressions at algorithmic level descriptions, it requires more run time and memory compared to our proposed hybrid representation (LTED) to represent RTL hardware designs that contain logical operations as well as arithmetic functions.

3 LTED as a Hybrid Bit and Word Levels Representation

In this section we will introduce a graph-based representation called LTED for functions with a mixed Boolean and integer domain and an integer range. Although LTED was introduced in [1] as a complete solution for formal property verification, in this paper we are going to evaluate the performance of LTED as a canonical graph-based representation in comparison with TED introduced in [2]. In order to have a canonical form, all nodes introduced in [1] except *Constant* (C) and *Variable* (V) nodes have been removed. In this representation basic arithmetic operators such as addition, subtraction, and multiplication are available that work for symbolic integer variables. In order to represent Boolean functions, logical bitwise operations including NOT, AND and OR have been provided. Fig. 1 depicts syntax of arithmetic and logical operations supported by LTED, where *Const*, *Var* and *term* denote a constant value, a Boolean or integer variable and word or bit level expression respectively. While arithmetic terms consist of subtraction ($\text{term}_1 - \text{term}_2$), addition ($\text{term}_1 + \text{term}_2$) and multiplication ($\text{term}_1 * \text{term}_2$), logical terms define negation (NOT term), conjunction (term_1 AND term_2) and disjunction (term_1 OR term_2) operations. Obviously, in contrast to HSAT [12], both operands of the multiplication operation can be integer variables.

```

description ::= list of terms
term ::= Const | Var | Var = term |
        arithmetic_terms | logical_terms
arithmetic_terms ::= term1 - term2 |
                    term1 + term2 | term1 * term2
logical_terms ::= NOT term | term1 AND term2 |
                term1 OR term2

```

Fig. 1. Arithmetic and Logical Operations in LTED.

This is important to be noted that in order to have a hybrid representation of Boolean and integer variables, the logical operations need to be supported as well as arithmetic expressions. As shown in Fig. 2(a), TED converts logical operations to arithmetic functions directly. In LTED, however, similar to BDDs, Shannon decomposition with respect to a Boolean variable is carried out in each Boolean node to have a canonical hybrid representation as illustrated in Fig. 2(b).

1) NOT X	==>	1 - X	(1-X) 1+ (X) 0
2) X AND Y	==>	X*Y	(1-X) 0+ (X) Y
3) X OR Y	==>	X+Y-X*Y	(1-X) Y+ (X) 1
		(a)	(b)

Fig. 2. Logical Operations (a) in TED, (b) in LTED

3.1 LTED Representation

In LTED, functions to be represented are maintained as a single graph in strongly 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. In contrast to TED, LTED is a binary graph-based representation where the algebraic expression $F(X, Y, \dots)$ is expressed by a first-order linearization of the Taylor series expansion [1]. Suppose variable X is the top variable of $F(X, Y, \dots)$. Equation (1) shows $F(X, Y, \dots)$, where $const$ is independent of variable X , while $linear$ is coefficient of variable X .

$$F(X, Y, \dots) = const + X * linear. \quad (1)$$

Definition: *LTED is a directed acyclic graph $G=(VR, ED)$ with vertex set VR and edge set ED . While the vertex set VR consists of two types of vertices: Constant (C) and Variable (V), the edge set indicates integer values as weight attribute. A Constant node v has as its attribute a value $val(v) \in Z$. A Variable node v has as attributes an integer variable $var(v)$ and two children $const(v)$ and $linear(v) \in \{V, C\}$.*

According to the above definition, Fig. 3 depicts vertex v in the LTED that denotes an integer function f^v defined recursively as follows:

- If $v \in C$ (is a Constant node), then $f^v = val(v)$.
- If $v \in V$ (is a Variable node), then $f^v = const(v) + var(v) * linear(v)$.

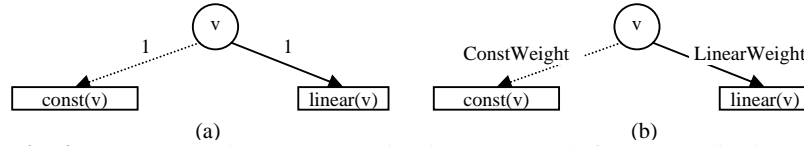


Fig. 3. Integer Function Representation in LTED (a) before Normalization (b) after Normalization

Although LTED supports polynomial functions by factoring higher orders of Taylor series expansion recursively as shown in Equation (2), we focus in this paper on the multivariate linear functions, leaving to another work the exploitation of the multivariate polynomial functions.

$$F(X, Y, \dots) = F(X = 0, Y, \dots) + x * [F'(X = 0, Y, \dots)] + \frac{1}{2!} x * [F''(X = 0, Y, \dots)] \cdot \quad (2)$$

3.2 LTED Operations

In this section, we express how to compose simple expressions to obtain complex ones. In order to do so, we describe how arithmetic operators such as addition and multiplication are applied to two LTED nodes and as a result a new LTED node is generated. Let u and v be two nodes to be composed, resulting in a new node q . Let

$\text{var}(u) = x$ and $\text{var}(v) = y$ denote the decomposing variables corresponding to the two nodes to be decomposed. The following cases should be considered:

- 1- If both nodes are *Constant* nodes ($u, v \in C$), a new *Constant* node q is computed as follows:

Addition	$q = u + v: \text{val}(q) = \text{val}(u) + \text{val}(v)$
Multiplication	$q = u * v: \text{val}(q) = \text{val}(u) * \text{val}(v)$

- 2- If one of the nodes is *Constant* node ($v \in C$), a new *Variable* node q is created as follows. For addition operation, the *const* part of result, i.e. q_0 , is obtained by adding constant $\text{val}(v)$ to *const* part of u (u_0). The *linear* part of result will be *linear* part of u (u_1). For multiplication operation, $\text{val}(v)$ should be multiplied with both *const* and *linear* parts of u (u_0 and u_1) to create *const* and *linear* parts of the result (q_0 and q_1) respectively.

Addition	$q = u + v: q_0 + x * q_1 = (u_0 + \text{val}(v)) + x * u_1$
Multiplication	$q = u * v: q_0 + x * q_1 = u_0 * \text{val}(v) + x * u_1 * \text{val}(v)$

- 3- If both nodes are *Variable* nodes ($u, v \in V$), proceed according to variable order. Suppose $\text{order}(x) > \text{order}(y)$.

- Where the two nodes are indexed by different variables, $\text{var}(q) = \max(\text{var}(u), \text{var}(v)) = x$. For addition operation, the *const* part of result, i.e. q_0 , is computed by adding v node to *const* part of u (u_0). The *linear* part of result will be *linear* part of u (u_1). For multiplication operation, v node should be multiplied with both *const* and *linear* parts of u (u_0 and u_1) to generate *const* and *linear* parts of the result (q_0 and q_1) respectively.

Addition	$q = u + v: q_0 + x * q_1 = (u_0 + v) + x * u_1$
Multiplication	$q = u * v: q_0 + x * q_1 = u_0 * v + x * u_1 * v$

- Where the nodes have the same index then $\text{var}(q) = x$. In this case, the *const* part of q is created by pairing the *const* parts of two nodes ($u_0 + v_0$ for addition and $u_0 * v_0$ for multiplication). The *linear* part of q is obtained as a sum of two cross products of *const* and *linear* parts when multiplication should be done. Furthermore, the quadratic term, i.e. q_2 , is taken into account in linear portion of q .

Addition	$q = u + v:$
	$q_0 + x * q_1 = (u_0 + v_0) + x * (u_1 + v_1)$
Multiplication	$q = u * v:$
	$q_0 + x * (q_1 + x * q_2) = u_0 * v_0 + x * (u_1 * v_0 + u_0 * v_1 + x * (u_1 * v_1))$

3.3 Reduction Rules and Canonicity

Analogous to BDDs and *BMDs, LTED can be reduced by removing redundant nodes and merging isomorphic nodes. In order to do so, the following reduction rules have been employed:

Rule 1: Remove a node if its *linear portion* (right child) is *Terminal 0* or its right edge has 0 weight, i.e. $LinearWeight=0$. Then replace this node with its *const portion* (left child). Fig. 4(a) illustrates this situation where node v contains only const part and therefore the function computed at that node is independent of variable $var(v)$.

Rule 2: Merge isomorphic nodes. This merging rule identifies isomorphic sub-graphs. Two nodes are isomorphic if they not only have the same *const* and *linear* portions but also their variable IDs should be the same. Fig. 4(b) shows that nodes $v1$ and $v2$ are isomorphic and then merge together as a new node v .

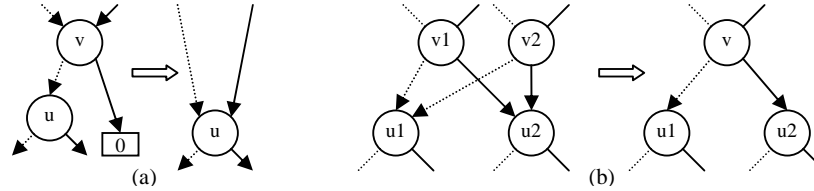


Fig. 4. Reduction Rules (a) Redundant Nodes (b) Isomorphic Sub-graphs

In order to further reduce the LTED graph, the numeric values from the *Constant* nodes are moved to the related edges and taken into account as edge *weight*. As a result, only *Constant* nodes 0 and 1 are needed in the graph. Furthermore, the weights can be propagated to the upper edges. This procedure is called *normalization* (see Fig. 3(b)). To do so, any common factor between the weights assigned to the edges of *const* and *linear* portions, is extracted by taking the greatest common divisor (*gcd*) of the argument weights. Once the weights have been normalized the hash table is looked 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 strongly canonical form due to merging isomorphic sub-graphs and removing redundant nodes.

3.4 Example

In order to have a better understanding of LTED representation, consider the following algebraic expression. Let the ordering of variables be $X > Y > Z$. Note that in our graph-based representation, dashed and solid lines indicate *const* and *linear* parts respectively.

$$f(X, Y, Z) = 24 \cdot 8 \cdot Z + 12 \cdot Y \cdot Z - 6 \cdot X \cdot Y - 6 \cdot X \cdot Y \cdot Z$$

First the decomposition with respect to variable X is taken into account. As illustrated in Fig. 5(a), after rewriting $f(X, Y, Z) = (24 \cdot 8Z + 12YZ) + X \cdot (-6Y - 6YZ)$ based on Equation (1), *const* and *linear* parts will be $24 \cdot 8Z + 12YZ$ and $-6Y - 6YZ$

respectively. After that, the decomposition is performed with respect to variable Y. By rewriting $24-8Z+12YZ = (24-8Z)+Y*(12Z)$, it is determined that the *const* and *linear* parts of the left hand side Y node in Fig. 5(b) are $24-8Z$ and $12Z$ respectively. While *const* and *linear* parts of the right hand side Y node are 0 and $-6-6Z$ respectively due to $-6Y-6YZ = (0)+Y*(-6-6Z)$. Finally the expressions are decomposed with respect to variable Z and a reduced diagram is depicted. In order to reduce the size of the LTED representation, 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. Fig. 5(c) depicts that $24-8Z$, $12Z$ and $-6-6Z$ are rewritten by $8[3+Z*(-1)]$, $12[0+Z*(1)]$ and $-6[1+Z*(1)]$ respectively. As mentioned above, in order to normalize the weights, $\gcd(8,12) = 4$ and $\gcd(0,6) = 6$ are taken to extract common factors as illustrated in Fig. 5(d). It should be noted that $\gcd(0,X)$ returns X. Finally, Fig. 5(e) shows the normalized graph where $\gcd(4,6) = 2$ is taken to extract common factor between out-going edges from X node.

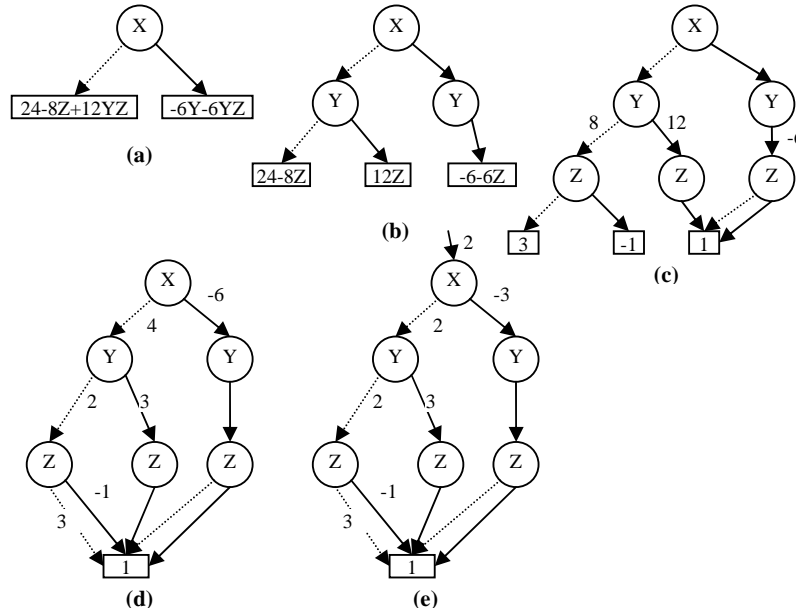


Fig. 5. LTED Representation of $24-8Z+12YZ-6XY-6XYZ$ (a) Decomposition with Respect to Variable X, (b) Decomposition with Respect to Variable X and Y, (c) Decomposition with Respect to Variable X, Y and Z, (d) Common Factor Extraction , (e) Normalized Representation

4 Experimental Results

In this section we will present our experimental results compared to TED package introduced in [2]. LTED package was implemented in C++ and has been carried out on an Intel 2.1GHz Core Duo and 1 GByte of main memory running Windows XP. In order to demonstrate that our package is applicable to industrial designs, experimental results of different benchmarks including 64-point Fast Fourier Transform (FFT64), Inverse fast Discrete Cosine Transform (IDCT), Differential Equation solver (DIFFEQ), MPEG2 [14] and Finite Impulse Response (FIR16) are presented and discussed.

General information about the benchmark circuits are given in Table 1. Column *benchmark* gives the benchmark's name, whereas in column *#lines* the number of lines of the specification (C code) after unrolling is provided. It is important to note that, all loops in the specification need to be unrolled to represent it in LTED or TED. The third, fourth and fifth columns (*#add*, *#sub* and *#mul*) provide the number of additions, subtractions and multiplications required in the C code of each benchmark respectively. We have used the topological order in which the signals appear in the C code.

It should be noted that BDDs and Boolean SAT based methods are not able to handle all benchmarks discussed here due to generating a large number of Boolean variables or clauses after encoding arithmetic functions into bit-level operations.

Table 1. Industrial Benchmark Characteristics.

Benchmark	#lines	#add	#sub	#mul
<i>FFT64</i>	1412	513	513	516
<i>IDCT</i>	690	312	224	288
<i>MPEG2</i>	340	128	104	144
<i>DIFFEQ</i>	90	20	20	30
<i>FIR16</i>	30	16	0	16

Each of these benchmarks is an important part of a system-level design where hardware and software parts are together described in a high level language like C, and therefore need to be co-validated. For example, one of our test-cases is 64-point FFT (FFT64) because high speed wireless data communication has found many application areas and fourth generation wireless and mobile systems are currently focusing on packet-based high-data-rate communication suitable for video transmission and mobile internet applications. Apart from the high speed of operation, the system demands lower power consumption due to above-mentioned applications. A general purpose DSP with associated software is not beneficial for this application since its power consumption is an order of magnitude higher compared to a dedicated hardware solution. In addition, it has been shown [15] through extensive simulation that the most computationally intensive parts of such a high-data-rate system are the 64-point inverse fast Fourier transform (IFFT) in the transmit direction and the

Viterbi decoder in the receive direction. Also in [16] people have implemented an 802.11a Transmitter and their synthesis results indicate that the IFFT needs 94.8% of total area and therefore is the most important part of the system. Automated formal tools, however, suffer from lack of a formal model to represent such a high-level description for property checking, high level synthesis and equivalence checking.

Although the FFT64 is realized by decomposing it into a two-dimensional structure of 8-point FFTs to reduce the number of required multiplications compared to the conventional radix-2 FFT64, we employ the conventional radix-2 FFT64 in order to have the maximum number of multiplications. Fig. 6 illustrates a pseudo code of FFT algorithm that carries out the butterfly computations with three main loops, where N is 64. The outside loop counts through the $\log_2(N)$ stages of the FFT computation and it causes huge data-dependent computations. The inner loops perform the individual butterfly computations of each stage. The heart of the FFT algorithm is the block of code that computes each butterfly in the third loop. In this figure, w_r and w_i parameters are commonly known as *twiddle factors* and can be computed before the algorithm is performed. But here we have taken into account them as symbolic variables rather than constant values to increase the number of variables and arithmetic operations.

```

for (s = 0 ; s < log2N ; s++)
  for (i = 0 ; i < N/(2s+1) ; i++)
    C = wr[idx]; S = wi[idx];
    for (j = i ; j < N ; j += N/2s)
      tmpr = aar[idx] - aar[idx+N/2s+1];
      tmpi = aai[idx] - aai[idx+N/2s+1];
      aar[idx] = aar[idx] + aar[idx+N/2s+1];
      aai[idx] = aai[idx] + aai[idx+N/2s+1];
      aar[idx+N/2s+1] = tmpr*C - tmpi*S;
      aai[idx+N/2s+1] = tmpr*S + tmpi*C;
    idx = idx + 2s;

```

Fig. 6. Pseudo Code of Fast Fourier Transform (FFT) Algorithm.

Table 2 summarizes the results of five benchmarks where the number of LTED (TED) nodes is given in column *#Nodes* and the number of input variables is reported in column *#Vars*. The memory and run time required to represent benchmarks in LTED or TED are provided in columns *Memory* (in MByte) and *CPU Time* respectively. All data concerning time is measured in seconds.

The run times for generation of LTED are almost two orders of magnitude lower than the run times of TED generation for all benchmarks. For example consider FFT64 benchmark that contains 190 input variables. In order to represent it in LTED or TED, 11220 nodes are needed, while the run time required to represent it in LTED and TED are 3 and 190 seconds respectively. The memory needed to generate TED is larger than that for LTED generation which are 15.7 MB and 10.8 MB for FFT64

benchmark respectively. For these benchmarks, two representations have the same number of nodes and edges so that theoretically it can be shown that both of TED and LTED packages require $O(2^{n-1})$ nodes and $O(2^n)$ edges, where n is the number of input variables. In spite of this fact, the LTED is faster than TED. We feel this difference is an effect of complex nodes and also normalization procedure utilized in TED. Since each node in TED has more than two edges, unlike to BDD or LTED, hash table in TED will have a complex structure with more than two children, so that this package wastes a lot of time looking for isomorphic sub-graphs and redundant nodes. It is interesting to note that, according to our previous experiment in [17] where TED was used to carry out equivalence checking between C-based specification and RTL implementation, run time required checking the equivalence of two descriptions of FFT64 test-case was 510, while run time needed to represent the specification of FFT64 benchmark in TED is 190 seconds. As a matter of fact, this comparison proves that TED package spends more than 60 percent of the run time to find out redundant nodes and isomorphic sub-graphs.

Table 2. Experimental Results.

BenchMark	#Nodes	#Vars	Memory (MByte)		CPU Time (Sec)	
			<i>LTED</i>	<i>TED</i>	<i>LTED</i>	<i>TED</i>
<i>FFT64</i>	11220	190	10.8	15.7	3	190
<i>IDCT</i>	3992	64	3.5	4.2	0.8	33
<i>MPEG2</i>	472	64	0.6	1.3	0.05	2
<i>DIFFEQ</i>	516	4	2.8	3.4	0.6	3
<i>FIR16</i>	10	9	0.1	0.7	0.002	0.5

Another type of experiments has been provided where Boolean variables are taken into account as well as integer variables. In order to evaluate the performance of two packages in a realistic environment, we consider bit-slicing of different integer variables in above-mentioned benchmarks and decompose them into shorter variables. We have divided experimental results into two categories (1) FFT64 experimental results after decomposition, (2) IDCT experimental results after decomposition.

4.1 FFT64 Benchmark

We have run two packages (LTED and TED) on 64-point Fast Fourier Transform (*FFT64*) benchmark, the most computationally intensive building block in communication systems, while some variables should be decomposed. To avoid rewriting the benchmark after decomposing of each variable, the decomposed variables have directly been changed into the net-list input file of two packages. In addition, we have considered power of two numbers of variables need to be decomposed according to three bit positions (1st, 5th and 10th bit positions) because it makes netlist modification easier.

Table 3 reports the experimental results where column $\#Decomp$ includes power of two numbers of decomposing variables that varies from 4 to 128. These variables have been decomposed based on three bit positions. For instance, last row indicates that 128 out of 190 total variables have been decomposed according to three bit positions (1st, 5th and 10th bit positions). These results allow comparison between LTED and TED for a real large enough industrial design. For different values of $\#Decomp$, the number of LTED or TED nodes ($\#Nodes$), the number of input variables ($\#Vars$), memory requirements in MByte ($Memory$) and run time needs in seconds ($CPU Time$) are summarized. In addition, the numbers of additions and multiplications have been tabulated in columns $\#add$ and $\#mul$ respectively to give a glimpse about the complexity of test-cases in terms of the number of arithmetic operations.

The results show the superiority of the LTED compared to TED. For a tightly restricted number of decomposed variables ($\#Decomp = 4, 8$ and 16) TED is feasible. But after that (for $\#Decomp = 32, 64$ and 128) the run time requirements grow prohibitively fast and TED fails. It could not handle this benchmark for $\#Decomp = 32$ after spending 40 minutes and using 52 MByte of memory. In contrast LTED exhibits a rather stable performance as run time and memory needs remain in the same order for all runs. Similar to previous experimental results, LTED is more than two orders of magnitude better than TED in terms of run time for $\#Decomp = 4, 8$ and 16 .

Table 3. FFT64 after Decomposition.

$\#Decomp$	$\#Nodes$	$\#Vars$	$\#add$	$\#mul$	Memory (MByte)		CPU Time (Sec)	
					LTED	TED	LTED	TED
4	14075	214	537	540	14	18	4	360
8	17289	238	561	564	20	24	5	480
16	23811	286	609	612	30	40	7.7	1020
32	37711	382	705	708	47	--	12	TO*
64	62530	574	897	900	76	--	18	TO
128	109046	958	1281	1284	130	--	27	TO

* TO = Time-Out

4.2 IDCT Benchmark

In the second category, we follow up on comparing LTED and TED by employing the data reported in our previous work where an equivalence checking approach based on TED has been proposed [17]. We have applied our approach on $IDCT$ benchmark when bit-slicing of a set of variables at four bit positions, i.e. 5th, 10th, 15th, and 20th, need to be done. Table 4 tabulates the results where the second and third rows indicate the number of nodes ($\#Nodes$) and the number of input variables ($\#InputVars$) respectively. The fourth and fifth rows give the number of additions ($\#add$) and multiplications ($\#mul$), while the two last rows compare LTED with TED

in terms of memory usage in MByte (column *Memory*) and CPU time in seconds (column *Time*).

Table 4. IDCT after Decomposition.

Cases		<i>case0</i>	<i>case1</i>	<i>case2</i>	<i>case3</i>	<i>case4</i>	<i>case5</i>	<i>case6</i>	<i>case7</i>
#Nodes		8120	12232	16368	20480	24560	28656	32768	36864
#InputVars		128	192	256	320	384	448	512	576
#add		376	440	504	568	632	696	760	824
#mul		352	416	480	544	608	672	736	800
Memory (MByte)	<i>LTED</i>	6.8	11.7	16.6	21	26.8	30.1	35.7	38.7
	<i>TED</i>	14	18.1	25	---	---	---	---	---
Time (Sec)	<i>LTED</i>	1.7	2.8	3.6	4.6	5.8	6.7	7.5	8.5
	<i>TED</i>	121	293	604	TO*	TO	TO	TO	TO
* TO: Time-Out									

In this table, eight cases are taken into account to apply bit-slicing to a set of input variables in *IDCT* test case. These cases are described as follows:

case₀: $b_{(8j)}$; for $j=0$ to 7

case _{i} : $b_{(i+8j)10}$ and case _{$i-1$} ; for $j=0$ to 7 and $i=1$ to 7

The *IDCT* benchmark has 64 input variables (b_0 - b_{63}) which are categorized into eight groups. For instance case₀ only comprises $b_0, b_8, b_{16}, b_{24}, b_{32}, b_{40}, b_{48}$ and b_{56} variables while case _{i} contains $b_1, b_9, b_{17}, b_{25}, b_{33}, b_{41}, b_{49}$ and b_{57} variables as well as variables included in case₀. Finally, case₇ consists of all 64 input variables. Each case was executed while bit-slicing at bit positions 5, 10, 15 and 20 (four bit positions) happened. For example, column *case1* provides experimental results where b_{8j} (i.e., $b_0, b_8, b_{16}, b_{24}, b_{32}, b_{40}, b_{48}$ and b_{56}) and b_{1+8j} (i.e., $b_1, b_9, b_{17}, b_{25}, b_{33}, b_{41}, b_{49}$ and b_{57}) must be decomposed into other variables according to the following equations:

$$b_{8j} = 2097152*b_{(8j)H4} + 1048576*b_{(8j)20} + 65536*b_{(8j)H3} + 32768*b_{(8j)15} + 2048*b_{(8j)H2} + 1024*b_{(8j)10} + 64*b_{(8j)H1} + 32*b_{(8j)05} + b_{(8j)L}$$

$$b_{1+8j} = 2097152*b_{(1+8j)H4} + 1048576*b_{(1+8j)20} + 65536*b_{(1+8j)H3} + 32768*b_{(1+8j)15} + 2048*b_{(1+8j)H2} + 1024*b_{(1+8j)10} + 64*b_{(1+8j)H1} + 32*b_{(1+8j)05} + b_{(1+8j)L}$$

Obviously, after increasing the number of decomposed variables to 32 variables (column *case3* in Table 4), *TED* package failed after about 40 minutes. In order to

compare two packages in terms of run time and memory usage, consider *case0* and *case2* where 8 and 24 variables should be decomposed respectively. Columns *case0* and *case2* in Table 4 tabulate the results after decomposing related input variables according to above-mentioned four bit positions. Although the number of nodes increases from 8120 for *case0* to 16368 for *case2*, memory needed in LTED grows from 6.8 MB to 16.6 MB. TED package, however, consumes 14 MB for *case0* and 25 MB for *case2*. Furthermore, run time required to represent the specification after decomposition increases from 1.7 to 3.6 seconds for LTED package, while TED package requires 121 seconds for *case0* and 604 seconds for *case2*. Once again the results prove that the CPU times required by LTED are almost two orders of magnitude larger than those of TED package due to this fact that TED spends more than 60 percent of the time to figure out isomorphic sub-graphs and redundant nodes.

5 Conclusion and Future Work

In this paper, we proposed a hybrid word-Boolean canonical representation namely LTED that is a suitable formal model for hardware/software co-validation. This representation is strong enough to handle arithmetic operations at the word level representation and there is no need to encode them as bit-level operations. In contrast to low level methods such as BDDs and Boolean SAT solvers, and even word-level representations such as TED, LTED is not only a compact and canonical form to represent symbolic expressions but also is very scalable even on large industrial circuits.

One possible avenue for future work would be to extend LTED to address modular arithmetic which is an important issue in hardware implementation. Furthermore, we are going to integrate LTED package in a SpecC-based environment in order to check the equivalence between different abstractions of SpecC as a system level language. Moreover, we are interested in applying LTED to high level synthesis applications to evaluate its performance even though polynomial functions need to be represented.

Acknowledgement

This work was partly supported by Semiconductor Technology Academic Research Center (STARC).

References

1. Alizadeh, B., Navabi, Z.: Word Level Symbolic Simulation in Processor Verification. In IEE Proceedings Computers and Digital Techniques Journal Vol. 151, No. 5 (2004) 356-366
2. Ciesielski, M., Kalla, P., Askar, S.: Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs. In IEEE Transactions on Computers Vol. 55, No. 9 (2006) 1188-1201
3. Bryant, R.: Graph-based Algorithms for Boolean Function Manipulation. In IEEE Transactions on Computers Vol. 35, No. 8 (1986) 677-691
4. Meinel, C., Sack, H.: \oplus -OBDD – a BDD Structure for Probabilistic Verification. In Workshop on Probabilistic Methods in Verification (1998) 141-151

5. Andersen, H., Hulgaard, H.: Boolean Expression Diagrams. In Logic in Computer Science (1997) 88-98
6. Horeth, S., Drechsler, R.: Formal Verification of Word-Level Specifications. In Proceedings of Design Automation and Test in Europe (DATE 1999) 52-58
7. Bryant, R.E., Chen, Y.-A.: Verification of Arithmetic Functions with Binary Moment Diagrams. In Proceedings of Design Automation Conference (DAC 1995) 535-541
8. Drechsler, R., Becher, B., Ruppertz, S.: The K*BMD: A Verification Data Structure. In IEEE Design and Test (1997) 51-59
9. Becker, B., Drechsler, R., Enders, R.: On the Representational Power of Bit-level and Word-level Decision Diagrams. In Proceedings of the Asia and South Pacific-Design Automation Conference (ASP-DAC 1997) 461-467
10. Stump, A., Barrett, C.W., Dill, D.L.: CVC: A Cooperating Validity Checker. In Proceedings of 14th International Conference on Computer Aided Verification (CAV 2002) 500-504
11. Moskewics, M., Madigan, C., Zhang, L., Zhao, Y., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In Proceedings of 38th Design Automation Conference (DAC 2001) 530-535
12. Fallah, F.: Coverage Directed Validation of Hardware Models. PhD thesis, MIT, 1999
13. Fallah, F., Devadas, S., Keutzer, K.: Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability. In Proceedings of 35th Design Automation Conference (DAC 1998) 528-533
14. MPEG Software Simulation Group: <http://www.mpeg.org/MSSG/>
15. Grass, E., Tittelbach, K., Jagdhold, U., Troya, A., Lippert, G., Krueger, O., Lehmann, J., Maharatna, K., Fiebig, N., Dombrowski, K., Kraemer, R., Aehoenen, A.: On the Single Chip Implementation of a Hiperlan/2 and IEEE802.11a Capable Modem. In IEEE Pers. Commun., Vol. 8 (2001) 48-57
16. Basha, E., Gerding, S., Liu, R.: Hardware Implementation of an 802.11a Transmitter. <http://web.mit.edu/rliu/www/publications/6884finalproject.pdf>
17. Alizadeh, B., Fujita, M.: A Hybrid Approach for Equivalence Checking Between System Level and RTL Descriptions. In 16th International Workshop on Logic and Synthesis (IWLS 2007) 298-304