Equivalence Checking of Integer Multipliers

Jiunn-Chern Chen
gis87519@cis.nctu.edu.tw

Yirng-An Chen
yachen@cis.nctu.edu.tw

Department of Computer & Information Science
National Chiao Tung University
Hsinchu, Taiwan 300, R. O. C.

Abstract— In this paper, we address on equivalence checking of integer multipliers, especially for the multipliers without structure similarity. Our approach is based on Hamaguchi's backward substitution method with the following improvements: (1) automatic identification of components to form proper cut points and thus dramatically improve the backward substitution process, (2) a layered-backward substitution algorithm to reduce the number of substitutions, and (3) Multiplicative Power Hybrid Decision Diagrams(*PHDDs) as our word-level representation rather than *BMD in Hamaguchi's approach. Experimental results show that our approach can efficiently check the equivalence of two integer multipliers. To verify the equivalence of a 32×32 array multiplier versus a 32×32 Wallace tree multiplier, our approach takes about 57 CPU seconds using 11 Mbytes, while Stanion's approach took 21027 seconds using 130 MBytes. We also show that the complexity of our approach is upper bounded by $O(n^4)$, where n is the word size, but our experimental results show that the complexity of our approach grows cubically $O(n^3)$.

I. Introduction

The practical motivation for study in this area is the high and increasing cost of correcting design errors in VLSI technologies. Design Errors or Bugs cost money, especially the hard-to-find bugs that surface late in the design cycle, that delay a product launch, or that require a massive product recall. For example, the Intel Pentium division bug [7] in 1994 demonstrated how important it is to catch errors early in the design cycle. This error cost Intel $475 million to recall the chips in the field. This bug was not covered by the one trillion test vectors used for this processor [6]. On May 10, 2000, a bug in Intel 820 chip set is reported and cost the company $200 millions of dollars to recall the chips [11]. These bugs demonstrated the inadequacy of simulation or emulation approaches to fully validate circuits, and the importance as well as the emergency need of formal verification approach to verify circuits.

Multipliers are widely used in current designs such as processors, digital signal processors (DSPs), graphic chip sets, etc. A bug in a chip with multiplier designs will cause the chips to be recall and thus cost a lot of money, similar to Intel's cases. Recently, designers are more careful to validate their new and improved designs by using equivalence checking tools to compare them against their reference designs. Many CAD vendors offer equivalence checking tools for design verification. For example, the popular equivalence checking tools are Formality from Synopsys, Tuxedo-LEC from Verplex and Design Verify from Avati. These tools performs logic equivalence checking of two circuits based the mixed approaches of functional and structural methods. In general, theses tools are very successful, even on multipliers, as long as the implementation circuits and the reference circuits have large structure similarity. Unfortunately, multipliers with different architectures do not provide enough structure similarity to allow verification using these tools. Based on our knowledge about these tools, they can not handle large complex arithmetic circuits with very little structural similarity such as multipliers with different design architecture. Thus, there is an emergence need to have an approach to check the equivalence of two multipliers.

Most of previous researches focused on verifying the correctness of integer multipliers based on Bit-level representations such as Binary Decision Diagrams (BDDs) [2] or word-level representations such as Multiplicative Binary Moment Diagrams (*BMDs) [4], etc. Bryant [3] has shown that the size of BDDs for multiplication grows exponentially with respect to the number of inputs regarding to any variable ordering. Yang *et. al* reported that the number of BDD nodes to represent integer multiplication grows exponential at a factor of about 2.87 per bit of word size [13]. For a 16-bit multiplier, building the BDDs for the output bits requires about 3.8GB memory on a 64-bit machine (i.e. 1.9GB on a 32-bit machine). Jain *et al* [9] have used Indexed Binary Decision Diagrams (IBDDs) to verify several multiplier circuits. They were able to verify C6288 (a 16-bit multiplier) in 22 minutes of CPU time. They were also able to verify a multiplier using Booth encoding, but this required almost 4 hours of CPU time and generated over 1 million vertices in the graphs. Stanion [12] has proposed a implicit verification method for verifying structurally dissimilar arithmetic circuits. Rather than trying to prove that two outputs are equivalent, they tried to find an implication of the form $A \rightarrow C$ where the consequent C states that the outputs are equivalent. The antecedent A is used to speed up the process of verifying C. The antecedent A will be verified recursively using the same manner. His experimental results show two 32-bit array and Wallace multipliers can be verified in 21027 seconds using

*This work was supported in part by Synopsys Inc. and the National Science Council, R. O.C., under contract no. NSC89-2215-E009-062.

0-7803-6634-4/2001/$10.00

130 MBytes.

The drawbacks of the bit-level approaches are the BDD explosion problem and the need of complex expected Boolean functions, which are difficult to be given. To overcome these drawbacks, Multiplicative Binary Moment Diagrams (*BMDs) [4] are proposed to provide a compact representation for these integer encodings and operations. A *BMD-based hierarchical approach was also proposed to verify integer multipliers. Using sub-specifications for subcomponents, this approach can handle extremely large multipliers with reasonable computational requirements. However, it requires users to partition the circuits into subcomponents. To overcome this constraint, Hamaguchi et al. [8] proposed a backward substitution method to construct *BMDs directly from circuit descriptions without any high level information. The main idea of their approach is to construct *BMDs starting from the primary outputs with assigned variables and then backwardly substitute the variable in the *BMDs with the function obtained from the circuits for each logic gate until the *BMDs only depend on the primary input variables. Based on their experimental results, the complexity of their approach grows $O(n^4)$ with respect to the word size n and it takes about 10263 seconds to complete the construction of *BMDs for a 64×64 multiplier.

In this paper, we address on equivalence checking of integer multipliers, especially for the multipliers without structure similarity. Our approach is based on Hamaguchi's backward substitution method [8] with the following improvements: (1) automatic identification of components to form proper cut points and thus dramatically improve the backward substitution process, (2) a layered-backward substitution algorithm to reduce the number of substitutions, and (3) Multiplicative Power Hybrid Decision Diagrams(*PHDDs) [5] as our word-level representation rather than *BMD in Hamaguchi's approach.

Experimental results show that our approach can efficiently check the equivalence of two integer multipliers. To verify the equivalence of a 32×32 array multiplier versus a 32×32 Wallace tree multiplier, our approach takes about 57 CPU seconds using 11 Mbytes, while Stanion's approach took 21027 seconds using 130 MBytes. We also show that the complexity of our approach is upper bounded by $O(n^4)$, where n is the word size, but our experimental results show that the complexity of our approach grows cubically $O(n^3)$.

II. Integer Multiplier Architecture

The integer multiplier is mainly divided into two portions: partial product generation to generate the partial product vectors for two operands A and B and sum of vectors to add up the partial product vector as shown in Figure 1. Usually, the partial products are generated by Bit-Pair or Booth Recoding Algorithm [1]. Given two operands A and B, where A is the multiplier word in the form: $A = a_{n-1}a_{n-2}...a_0$, and B is the multiplicand in the form: $B = b_{n-1}b_{n-2}...b_0$, Bit-Pair approach generates n n-bit vectors as follows: $\vec{p_i} = [a_i \cdot b_{n-1}, a_i \cdot b_{n-2}, \ldots, a_i \cdot b_0], 0 \leq i \leq n-1$. The second partial product generation approach is the Booth Recoding Algorithm [1]. Based on the radix number, this technique groups several bits in the operand A at a time together with another operand B to generate one partial product vector. To generate all partial product vectors, the operand A will be partitioned into several groups and adjacent groups share one common bit.

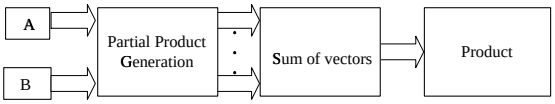

Fig. 1. Multiplier Architecture.

After the partial product vectors are generated, they are added together to form the total product based on the arrangement of the adders. There are many variation approaches to perform the summation of partial product vectors. In these approaches, there are component components, 1-bit full adder and half adder, to be used in this part of circuits. Two common approaches are array multiplier and Wallace tree.

III. *PHDD Preliminary

Multiplicative Power Hybrid Decision Diagrams (*PHDDs) [5] is proposed to represent functions that map Boolean vectors to integer or floating point values. *PHDDs use three decompositions as expressed in the following Equation 1:

$$\langle w, f \rangle = \begin{cases} c^w \cdot (((1-x) \cdot f_{\overline{x}} + x \cdot f_x)(Shannon) \\ c^w \cdot (f_{\overline{x}} + x \cdot f_{\delta x})(Positive\ Davio) \\ c^w \cdot (f_x + (1-x) \cdot f_{\delta \overline{x}})(Negative\ Davio) \end{cases}$$

where $< w, f >$ denotes $c^w \times f$ and $f_{\delta \overline{x}} = f_{\overline{x}} - f_x$ is the partial derivative of f with respect to $\overline{x}$. In general, c can be any integer. To verify arithmetic circuit, c is 2, because the base value of most arithmetic circuits is 2.

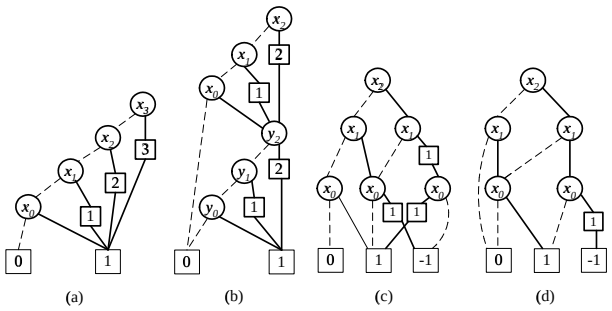

Fig. 2. *PHDD representations of four functions: (a) X, (b) $X \times Y$, (c) $sum(x_2, x_1, x_0)$ and (d) $carry(x_2, x_1, x_0)$.

*PHDDs can represent integer functions efficiently. For example, Figure 2.(a) and (b) show $\sum_{i=0}^{3} 2^i x_i$ and $X * Y = \sum_{i=0}^{2} 2^i x_i \times \sum_{i=0}^{2} 2^i y_i$, respectively. Edge weight i in *PHDDs represents 2^i and unlabeled edges have weight 0(2^0). The edge weights combine multiplicatively. The dash (solid) line represents the 0(1)-branch. *PHDDs grows linearly with the word size for the different encoding functions and integer multiplication, and can also represent Boolean functions efficiently. Figure 2.(c) and (d) represents the sum and carry functions of 1-bit full adder with inputs x_2, x_1 and x_0, respectively.

In [5], experimental results showed that *PHDD is at least five times faster than *BMDs to verify integer multipliers. Readers refer to) [5] for more details of *PHDD, due to the space limitation.

IV. Our Verification Approach

Let us formally define the problem: Given two integer multipliers with different architectures in gate-level netlist format, the task is to check whether these two circuits are equivalent.

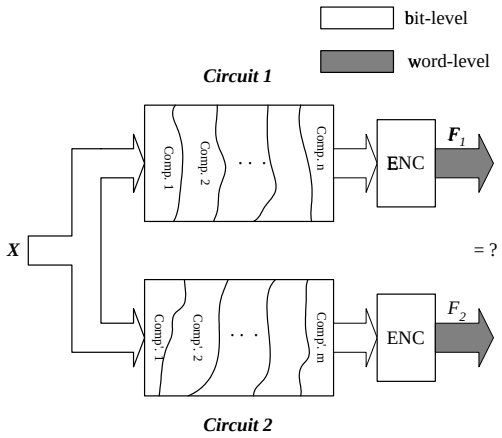

Fig. 3. A sketch of our approach.

Figure 3 illustrates schematically our approach to verify two multiplier circuits whether they are equivalent. First, for each circuit, an component identification algorithm is perform to automatically find expected components such as Booth recoding, 1-bit adder, etc. Then, these components will be sorted to form several cut lines according to their connectivity. Thus, the circuit is partitioned into several portions by these cut lines. After that, each signal in these cut lines, including the primary outputs, are assigned to a BDD variable. Based on the encoding, *PHDDs is constructed to represent the word-level function F of primary outputs in terms of output variables. Then, Layered-backward substitution algorithm is applied on F recursively from the outputs toward the inputs, to construct the *PHDDs in term of input variables only. The above procedure is applied twice, one on implementation circuit and another on the reference circuits, to get two *PHDDs. Thus, the equivalence of two circuits can be checked by comparing whether two *PHDDs are equal or not. If not, a counterexample is generated to point out the difference of two circuits.

A. Component Identification

After reading and flattening the circuits, the first step is to perform component identification and mark the outputs of these component as cut points. Based on our observation, the proper cut points are the output of partial product generation and the outputs of 1-bit adders. Thus, the components we want to identify are the partial product generation and 1-bit adder cells in the sum-of-vectors part of integer multipliers.

In order to identify a component whether it implements the expected functions, we incorporate a BDD symbolic simulator to simulate circuit and then compare the BDDs in the circuit with the expected functions to identify the components we want. First, BDD variables are assigned for specific inputs, which are the input candidates of the expected component. Next, symbolic simulation is performed through the circuit using these BDD variables. For expected functions, their BDDs are also built on the same BDD variables. After both BDDs are built, the expected BDDs are used to check whether there exists some signals whose BDDs are the same as the expected. If so, we find the expected component and these signals will be identified and marked as cut points.

Algorithm: Comp_Ident(η)
Input: η—A gate-level combinational circuit;
Output: cpQ—identified cut points;
1 $A \leftarrow multplicand$; $B \leftarrow multiplicator$;
/* identify partial products(pQ) */
2 $pQ \leftarrow BoothR4(A, B, \eta)$;
3 if !(pQ) $pQ \leftarrow BoothR2(A, B, \eta)$;
4 if !(pQ) $pQ \leftarrow BitPair(A, B, \eta)$;
5 if !(pQ) return pQ;
6 $cpQ \leftarrow pQ$;
7 while adders exist /* identify full or half adders */
8 if ($Adder \leftarrow check\_FullAdder(pQ, \eta)$) then
9 else ($Adder \leftarrow check\_HalfAdder(pQ, \eta)$);
10 if ($Adder$) then
11 $I \leftarrow$ inputs of $Adder$;
13 $O \leftarrow$ outputs of $Adder$;
14 $pQ \leftarrow pQ - \{I\} \cup \{O\}$;
15 $cpQ \leftarrow cpQ \cup \{O\}$;
16 Sorting cpQ by levels;
17 return cpQ

Fig. 4. Our component identification algorithm

Figure 4 show our algorithm to identify the components we want. First, we identify the components of partial product generation, according to the partial product rules. Currently, we implemented three algorithms, $BoothR4$, $BoothR2$ and $BitPair$, to identify three types of partial product generation circuits, Booth recoding based on Radix-4, Booth recoding based on Radix-2 and Bit-Pair, respectively. Procedure $BoothR4$ is called to detect whether the circuit use Booth Radix-4 recoding method to generate partial products and returns the set of cut points, if detected. If it is not based on Booth Radix-4 recoding method, procedure $BoothR2$ and $BitPair$ are applied in the order. If these three algorithms can not detect the partial product generations, the empty set is returned and the verification process will be aborted.

After the partial products are identified, the next task is to identify 1-bit full and half adders, shown in lines 7 to 15 in Figure 4. It first try to identify 1-bit full adder, and then try to identify 1-bit half adder. If both are not found, it exits the while loop; Otherwise, it deletes the inputs of the adder from the cut point set pQ, and adds the outputs to the sets pQ and cpQ. After the exit of the while loop, set cpQ contains all of the cut points we want. Then, cpQ will be sorted, according to their levels from the primary outputs to the primary inputs. Finally, the procedure returns the sorted cut point set cpQ.

B. Layered-Backward Substitution Approach

After the proper cut lines are found, we construct the overall function of the circuit by the layered-backward substitution method using *PHDDs as our word-level representation. To perform backward substitution method for constructing *PHDDs of integer multipliers, variables $z_m, z_{m-1}, \ldots, z_0$ are assigned to the outputs of integer multiplier. Based on the given encoding (e.g., unsigned encoding), *PHDDs for word-level function F ($\sum_{i=0}^{m} 2^i \times z_i$) is constructed in terms of these variables.

Algorithm: LB_Subst(L,P,F)
Input: L—Number of levels;
P–Sets of cut points;
F–*PHDDs in terms of output variables.
Output: $Result$–*PHDDs of the circuit in terms of input variables;
1 $Result \leftarrow F$;
2 for (i=$L-1$; $i >= 0$; i++)
3 $C_2 \leftarrow$ set of cut points at level $i+1$;
4 $C_1 \leftarrow$ set of cut points at level i;
5 assign variables to each cut point in C_1;
6 Simulate(C_1, C_2);
7 for each cut point p in C_2
8 get variable f of p;
9 get the evaluated function g of p from the simulation;
10 $Result \leftarrow Substitute(Result, f, g)$;
11 return $Result$;

Fig. 5. The algorithm of Layered-Backward Substitution.

Then, algorithm LB_Subst, shown in Figure 5, is applied to perform backward substitution on word-level function F. Assume that the circuits have L+1 levels of cut points, where level 0 is the primary inputs, level 1 is the partial productions and level L is the primary outputs. The algorithm repeats the following process L times starting from level L-1. First, it gets the sets of cut points, C_1 and C_2, at level i and $i+1$, respectively. BDD variables are assigned to the cut points in C_1. After that, a symbolic simulator is called to simulate the circuits with BDD variables in C_1 as inputs and build BDD functions for the cut points in C_2. Now, every cut point in C_2 is associated to a variable f and a Boolean function g, which is a function of BDD variables in C_1. Since $Result$ is depended on variable f, variable f in $Result$ is substituted by Boolean function g, for every cut point in C_2. After repeating the above process L times, function $Result$ depends on input variables only. Observe that only two sets of variables are needed simultanously in the substitution process. Thus, BDDs variables can be reused.

V. Complexity of Our Approach

To analyze the complexity of our approach, we begin with the complexity of $Comp\_Ident$ algorithm and then the complexity of layered backward substitution algorithm.

Theorem 4.1 The complexity of $Comp\_Ident$ algorithm is bounded by $O(n^3)$, where n is the word size.

Proof: The algorithm is composed of partial product and adder cell identification. For the partial product identification, there are at most $O(n^2)$ products (i.e., using BitPair approach) to be identified. The identification process of each product takes contant time. Thus, the complexity of partial product identification is bounded by $O(n^2)$.

For the adder cell identification, there will be $O(n^2)$ adders to add $O(n^2)$ partial products into the final result. The identification process of an adder cell takes constant time. Thus, the complexity of adder cell identification is bounded by $O(n^2)$. Since there are $O(n^2)$ cut points, sorting them according to their levels are bounded by $O(n^3)$. Therefore, the complexity of $Comp\_Ident$ algorithm is bounded by $O(n^3)$.□

To represent encoding functions, *PHDD is a specific structure, called *Sum Of weighted Variables* (SOV) [10]. For example, *PHDD for unsigned encoding function $f = \sum_{i=0}^{n-1} a_i \times 2^i$, shown in Figure 2.(a) with n=4, is a SOV-structure. By extending the Lemma 4.1 and 4.2 in [10] for *BMDs, we analyze the costs of constructing the *PHDD for the adder-part of the multiplier. First, we analyze the substitution costs for a single full adder to get the overall costs.

Lemma 4.1 Let F be a *PHDD and let X denotes the set of variable of F. Let x_i, x_j be two variables in X, representing the sum- and carry-output of the same full adder. The inputs to the full adder are represented by $x_k, x_l, x_m \notin X$. Independent of the order on the variable sets of the *PHDDs involved in the substitution process, it holds that substituting x_i, x_j in F by the *PHDDs for the functions $sum(x_k, x_l, x_m)$, $carry(x_k, x_l, x_m)$ generates a *PHDD F' with variable set $(X - \{x_i, x_j\}) \cup \{x_k, x_l, x_m\}$ and:

1. F' is in SOV.
2. The terminal value of the high-successors of the nodes for x_k, x_l, x_m in F' is the same as that for the high-successor of the node for x_i in F.

Proof: For the sum- and carry-output based on functional argument, we get the following functions:

$$\begin{aligned} sum(x_k, x_l, x_m) &= x_k \oplus x_l \oplus x_m \\ &= x_k + x_l + x_m - 2x_kx_l - 2x_kx_m \\ &\quad -2x_lx_m + 4x_kx_lx_m \\ carry(x_k, x_l, x_m) &= (x_k \wedge x_l) \vee (x_k \wedge x_m) \vee (x_l \wedge x_m) \\ &= x_kx_l + x_kx_m + x_lx_m - 2x_kx_lx_m \end{aligned}$$

by expressing the boolean operations $\oplus$ and $\wedge$, $\vee$ by integer addition, subtraction and multiplication, i.e., $x \oplus y = x + y - 2xy$, $x \wedge y = xy$, $x \vee y = x + y - xy$.

Assume x_i and x_j variables have weights 2^w and 2^{w+1}, respectively. The substitution of x_i, x_j in f by $sum(x_k, x_l, x_m)$, $carry(x_k, x_l, x_m)$ yields the following function F'.

$$\begin{aligned} F' &= \ldots + 2^w \cdot sum(x_k, x_l, x_m) \\ &\quad +2^{w+1} \cdot carry(x_k, x_l, x_m) + \ldots \\ &= \ldots + 2^w \cdot x_k + 2^w \cdot x_l + 2^w \cdot x_m + \ldots \end{aligned}$$

as can easily be verified. Since $x_k, x_l, x_m \notin X$ and the weighted variables x_i, x_j appear only once in F, the rest of the variables in F are not affected. Therefore the *PHDD of F' must

be in SOV again. Furthermore all three high-edges of the nodes for x_k, x_l, x_m in F' point to an edge weight node with value w.□

With Lemma 4.1 we can be sure the SOV-structure of the *PHDD is maintained, if we substitute the variables corresponding to both outputs of a particular full adder directly one after another. Furthermore, at each point of the substitution process, the size of *PHDD is increased by one after having processed the next full adder.

Lemma 4.2 Let F be a *PHDD in SOV with size $|F|$. Let $X, x_i, x_j, x_k, x_l, x_m, sum(x_k, x_l, x_m)$ and $carry(x_k, x_l, x_m)$ be defined as in Lemma 4.1. Substituting x_i, x_j in F by $sum(x_k, x_l, x_m)$ and $carry(x_k, x_l, x_m)$ is bounded by $O(|F|)$ with respect to time, independent of the variable order or order of substitutions.

Proof: We denote the *PHDDs for $sum(x_k, x_l, x_m)$ and $carry(x_k, x_l, x_m)$ by S and C. For considering first the substitution of x_i by S, the substitution process can visualized in Figure 6. The substitute algorithm calls itself recursively until it reaches the node in F labeled with x_i. Obviously, the number of recursive calls is bounded by $O(|F|)$. At the node labeled with x_i, the operation $F_{low(x_i)} + S \cdot F_{high(x_i)}$ has to be carried out, where $F_{low(x_i)}, F_{high(x_i)}$ denote the *PHDDs to which the low- and high-edge of node x_i point. Since F is in SOV, $F_{high(x_i)}$ is a terminal node and the call to multiplication operation ends immediately. Since $F_{low(x_i)}$ is in SOV, too, and S is of constant size, the addition is bounded by $O(|F|)$. The *PHDD after this substitution can be seen in Figure 7 for a variable order $x_k < \ldots < x_l < \ldots < x_m$.

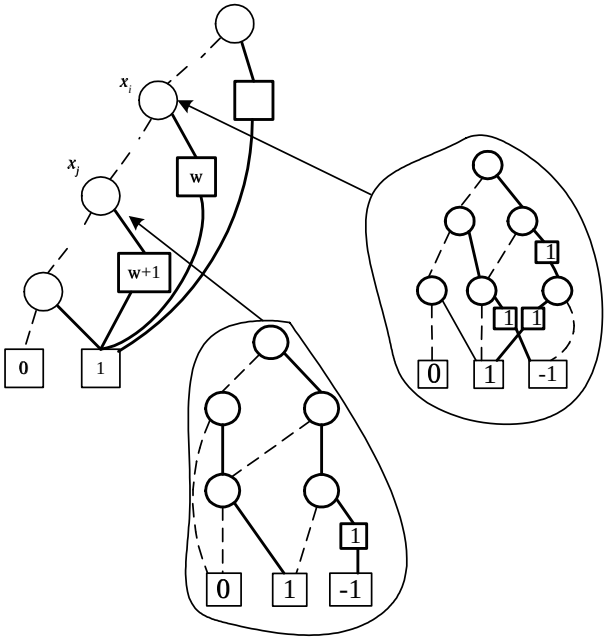

Fig. 6. Representation of substituting x_i by S and x_j by C.

For the substitution of x_j by C, analogous arguments hold. The *PHDD after this substitution can be seen in Figure 7.b for a variable order $x_k < \ldots < x_l < \ldots < x_m$. Therefore, since the resulting *PHDD is of size $|F| + 1$ according to Lemma 4.1, we get a time bound of $O(|F|)$ for the substitution of x_i by S and x_j by C and $(X - \{x_i, x_j\}) \cup \{x_k, x_l, x_m\}$.□

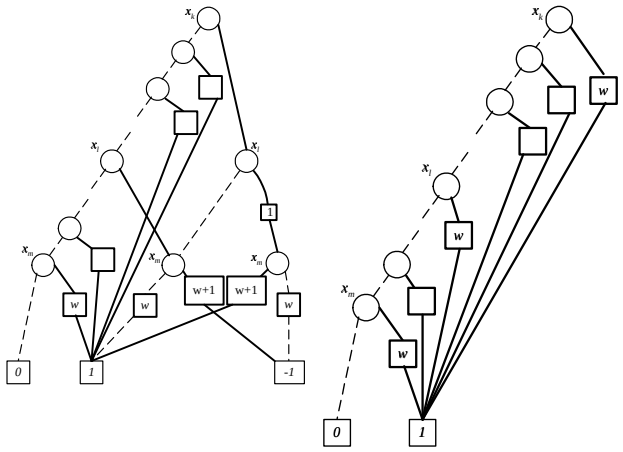

Fig. 7. (a) *PHDD after substitution of variable x_i only. (b) *PHDD after substitution of variable x_i and x_j.

Theorem 4.2 Constructing the *PHDD for the adder-part using substitution is bounded by $O(n^4)$.

Proof: For the proof we first count the number of adder components. Depending on the realization, the exact number of these elements differs. Asymptotically there are $m_1 = O(n^2)$ adders forming the adder-part of any multiplier. Therefore the number of execution steps has an upper bound of $\sum_{i=0}^{m_1-1}(|F_0| + i) = O(n^4)$, where $|F_0| = O(n)$ is the size of the initial *PHDD.□

We analyzed the complexity of layered-backward substitution for the part of the multiplier circuit that adds the partial products to obtain the result of the multiplication above. In the sequence we analyze the costs of substituting the partial products into the *PHDD. Our starting point is the *PHDD constructed up to the outputs of the AND gates. It has size $m' = O(n^2)$, since there are n^2 partial product bits.

Theorem 4.3 Let F be the *PHDD constructed for the adder part. Substitution of the variables of F by the *PHDDs for partial products $a_i \cdot b_j$ is bounded by $O(n^4)$ with respect to time with component sorting for the AND gates.

Proof: Since the number of substitutions for partial products is bounded by $O(n^2)$ and the size of the *PHDD F after substituting the part of summing up partial products is $O(n^2)$, we get an overall time bound of $\sum_{i=1}^{m'} O(n^2) = O(n^4)$.□

Combining Theorem 4.1, 4.2 and 4.3, we conclude that the complexity of our approach is bounded by $O(n^4)$.

VI. Experimental Results

We have implemented our approach in C using CUDD and *PHDD packages. To check the equivalence of two multipliers, both circuits pass same path from parsing and flattening procedure to layered-backward substitution procedure and then compare against each other. If they are equivalent, then it

report that implementation circuit is correct; Otherwise it generates counterexamples for modifying the differences between them.

Bits	Mult A		Mult B		Mult C	
	CPU	MEM	CPU	MEM	CPU	MEM
4	0.1	6.2	0.1	6.1	0.2	7.7
8	0.6	6.9	0.6	6.9	0.5	7.9
16	4.6	8.2	3.0	8.5	2.1	8.3
32	35.4	9.1	21.5	9.9	10.4	9.5
64	284.1	12.2	213.5	16.3	89.5	14.6
order	$n^{2.8}$	$n^{1.1}$	$n^{3.1}$	$n^{1.3}$	$n^{2.9}$	$n^{1.2}$

TABLE I RESULTS OF THREE DIFFERENT TYPES OF MULTIPLIERS.

We first measure the efficiency of our backward *PHDD construction for different integer multipliers. Our experiments are performed on Sun UltraSparc60 (450MHz). Table I shows that the CPU time (second) and used memory (MBytes) for the construction of *PHDDs of multiplier circuits sized from 4-bit to 64-bit using different architectures. Mult A is based on bit-pair and array multiplier. Mult B is based on Bit-Pair and Wallace-tree multiplier. Mult C is based on Booth-Radix4 and Wallace-tree Multiplier. Note that its CPU time grows near cubically with the word size and its memory grows linearly with the word size. Compared with Hamaguchi's backward substitution [8], our approach took 284 seconds for a 64-bit multiplier, while their approach took 10263 seconds. Note that the CPU time of their approach is bounded by $O(n^4)$ and ours is bounded by $O(n^3)$.

Bits	CUP(Sec)	MEM(MB)
4	0.2	7.7
8	1.1	8.1
16	7.5	8.6
32	56.7	10.7
64	526.3	19.5

TABLE II EQUIVALENCE CHECKING OF TWO MULTIPLIERS

Our system is applied to verify the equivalence of two multipliers using different architectures. Table II shows our experimental results for the equivalence checking of array and Wallace tree multipliers based on same Bit-Pair partial product generation. From Table II, we can see that the amount memory required tripled at most for each double bits in the multipliers. Moreover, the CPU time is approximately that we added up the times for constructing *PHDDs of two circuits separately. Compared with Stanion's approach [12], we have been able to verify a 32×32 array multiplier versus a 32×32 Wallace tree multiplier in 57 seconds, while his approach took about 21027 seconds. Note that our approach can verify 64-bit multipliers in 527 seconds.

VII. CONCLUSIONS

We addressed on equivalence checking of integer multipliers, especially for the multipliers without structure similarity. We presented an approach based on Hamaguchi's backward substitution method with the following improvements: (1) automatic identification of components to form proper cut points, (2) a layered-backward substitution algorithm, and (3) Multiplicative Power Hybrid Decision Diagrams as our word-level representation. Experimental results show that our approach can efficiently check the equivalence of two integer multipliers. We show that the complexity of our approach is upper bounded by $O(n^4)$, where n is the word size. Our experimental results show that the complexity of our approach grows cubically $O(n^3)$. In the future, we would like to our approach to verify other arithmetic circuits such as $A \times B + C$.

REFERENCES

[1] A. D. Booth. A signed binary multiplication technique. In *Journal of Mechanics and Applied Mathematics*, pages 236–240, 1951.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, pages 8:677–691, August 1986.

[3] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. In *IEEE Transactions on Computers*, pages 2:205–213, Feb 1991.

[4] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 535–541, June 1995.

[5] Y.-A. Chen and R. E. Bryant. *PHDD: An efficient graph representation for floating point circuit verification. In *Proceedings of the International Conference on Computer-Aided Design*, pages 2–7, November 1997.

[6] Y.-A. Chen, E. M. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In *Proceedings of the Formal Methods on Computer-Aided Design*, pages 19–33, November 1996.

[7] T. Coe. Inside the Pentium Fdiv bug. *Dr. Dobbs Journal*, pages pp. 129–135, April 1996.

[8] K. Hamaguchi, A. Morita, and S. Yajima. Efficient construction of binary moment diagrams for verifying arithmetic circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 78–82, November 1995.

[9] J. Jain, J. Bitner, M. S. Abadir, J. A. Abraham, and D. S. Fussell. Indexed BDDs: Algorithmic advances in techniques to represent and verify boolean functions. In *IEEE Transactions on Computers*, pages 11:1230–1245, November 1997.

[10] M. Keim, M. Martin, R. Drechsler, and P. Moliter. Polynomail formal verification of multipliers. In *Proceedings of 15th IEEE VLSI Test Symposium*, pages 150–155, 1997.

[11] T. Mainelli. Intel 820 owners could get free upgrade. In *PC World*, May 2000.

[12] T. Stanion. Implicit verification of structurally dissimilar arithmetic circuits. In *Proceedings of 1999 IEEE International Conference on Computer Design: VLSI in Computer and Processors*, pages 46–50, October 1999.

[13] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallaron. Space- and time-efficient bdd construction via working set control. In *Proceedings of ASP-DAC '98*, pages 423–432, Yokohoma,Japan, Feb. 1998.