PAPER *Special Section on VLSI Design and CAD Algorithms*

An Efficient Method for Finding an Optimal Bi-Decomposition

Shigeru Yamashita[†], *Nonmember*, **Hiroshi Sawada**[†], *and* **Akira Nagoya**[†], *Members*

SUMMARY This paper presents a new efficient method for finding an "optimal" bi-decomposition form of a logic function. A bi-decomposition form of a logic function is the form: $f(X) = \alpha(g_1(X^1), g_2(X^2))$. We call a bi-decomposition form optimal when the total number of variables in X^1 and X^2 is the smallest among all bi-decomposition forms of f. This meaning of optimal is adequate especially for the synthesis of LUT (Look-Up Table) networks where the number of function inputs is important for the implementation. In our method, we consider only two bi-decomposition forms; $(g_1 \cdot g_2)$ and $(g_1 \oplus g_2)$. We can easily find all the other types of bi-decomposition forms from the above two decomposition forms. Our method efficiently finds one of the existing optimal bi-decomposition forms based on a branch-and-bound algorithm. Moreover, our method can also decompose incompletely specified functions. Experimental results show that we can construct better networks by using optimal bi-decompositions than by using conventional decompositions.

key words: functional decomposition, bi-decomposition, AND, XOR, look-up table

1. Introduction

When implementing a combinational logic function using a given technology, the desired function must be decomposed or factorized to smaller functions so that the decomposed functions can fit onto the implementation primitives of the technology. Many methods have been proposed to decompose functions. Among such methods, AND/OR factoring and weak division [1] are superior when expressions are in sum-of-products forms. However, other approaches produce better results in some cases. For example, factoring with XOR can express some logic functions simpler than AND/OR factoring [2], [3]. For the synthesis of LUT (Look-Up Table) networks, functional decomposition [4] based methods can often produce better results [5].

Most of the previously proposed functional decomposition methods have been based on Roth-Karp decomposition [6], and thus they decompose function f to the following form: $f = \alpha(g_1(X^B), \ldots, g_t(X^B), X^F) = \alpha(\vec{g}(X^B), X^F)$, where X^B and X^F are variable sets. We can think of another strategy for functional decomposition: function f is decomposed into only two functions as $f = \alpha(g_1(X^1), g_2(X^2))$. This decomposition is called bi-decomposition [7]. Since bi-decomposition and Roth-Karp decomposition forms are very different, bi-decomposition is useful for some functions. If X^1 and X^2 are limited to disjoint sets, the bi-decomposition form can be found very quickly [7]. In some cases, only a "non-disjoint" bi-decomposition form can provide the best decomposition (an example is shown in Section 2). Therefore, we need an efficient method for finding non-disjoint bi-decomposition forms.

The methods proposed in [8], [9] can find non-disjoint bi-decomposition forms using the notion of "groupability". They can find a bi-decomposition form for given X^1 and X^2, but they have a problem in selecting the best X^1 and X^2.

In this paper, we propose an efficient method for finding an "optimal" non-disjoint bi-decomposition form of an incompletely specified function. Here, "optimal" means that the total number of variables in X^1 and X^2 is the smallest among all bi-decomposition forms. This meaning is thought to be adequate for the synthesis of LUT networks, because an LUT can realize a complex function if the number of input variables does not exceed the maximum number of inputs of the LUT. Our method can provide a solution to the problem of selecting the best X^1 and X^2 in a bi-decomposition form, especially in LUT network synthesis.

This paper is organized as follows. In Section 2, we explain non-disjoint bi-decomposition and formulate our problem. In Section 3, we present an overview of our strategy for the problem. We thoroughly explain the techniques used in our strategy in Sections 4 and 5. We present experimental results in Section 6. We conclude the paper in Section 7.

2. Preliminaries

2.1 Non-Disjoint Bi-Decomposition

The decomposition form, $f = \alpha(g_1(X^1), g_2(X^2))$, is called a bi-decomposition form [7]. If X^1 and X^2 are disjoint, it is called a "disjoint" bi-decomposition form; if X^1 and X^2 are not disjoint, it is called a "non-disjoint" bi-decomposition form.

Disjoint bi-decomposition forms are very useful for logic synthesis and they can be quickly found [7]. However, there are functions that can be efficiently decomposed only by non-disjoint bi-decomposition. For example, suppose we want to decompose $(x_1+x_2+x_3)\cdot(x_2 \oplus$

Manuscript received March 13, 1998.
Manuscript revised June 15, 1998.

[†]The authors are with NTT Communication Science Laboratories, 2-4 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0237, JAPAN

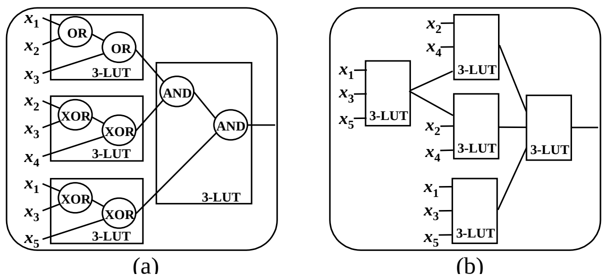

Fig. 1 LUT Networks based on Non-Disjoint Bi-Decomposition (a) and Roth-Karp Decomposition (b)

$x_3 \oplus x_4) \cdot (x_1 \oplus x_3 \oplus x_5)$. Although we cannot decompose the function by using disjoint bi-decomposition, we can find a good decomposition form as shown in Fig. 1 (a) by recursively using non-disjoint bi-decomposition. If we want to realize the function by 3-input LUTs, we can get an LUT network as shown in Fig. 1 (a). This network is the same as a straightforward realization from the expression. With a Roth-Karp decomposition based method [5], we cannot find such a good decomposition form for this example and unfortunately we get a worse network as shown in Fig. 1 (b). (The internal logics of LUTs are too complex to be expressed in the figure.)

2.2 Problem Formulation

In LUT network synthesis, one of the costs of a function is the number of variables which the function depends on. Therefore, we define an "**optimal**" bi-decomposition form as follows:

Definition 1: $f(X) = \alpha(g_1(X^1), g_2(X^2))$ is called an "**optimal**" bi-decomposition form if the total number of variables in X^1 and X^2 is the smallest among all bi-decomposition forms of f. □

If X^1 (or X^2) is an empty set, the decomposition is called a *trivial* decomposition, which we ignore in this paper. For example, $f(X) = 1 \cdot f(X)$ is a trivial decomposition, and X^1 is an empty set. A trivial decomposition is *not* called optimal in this paper. Therefore, there may be no optimal bi-decomposition forms for some functions which have only trivial decompositions. Note that an optimal bi-decomposition of a function may be disjoint or non-disjoint depending on the function.

The goal of this paper is to find an optimal bi-decomposition form for an incompletely specified function. Although there may be more than one optimal bi-decomposition form according to our definition, our problem is to find just one of them.

3. Overview of Our Strategy

3.1 The Whole Strategy of Our Method

Our strategy to find an optimal bi-decomposition form for f selects the best one among the following three types of bi-decomposition forms:

- an optimal bi-decomposition whose form is ($f = g_1 \cdot g_2$) which we call an "*AND-Decomposition*" form.
- an optimal bi-decomposition whose form is ($f = g_1 + g_2$) which we call an "*OR-Decomposition*" form.
- an optimal bi-decomposition whose form is ($f = g_1 \oplus g_2$) which we call an "*XOR-Decomposition*" form.

Although we can think of other types of bi-decomposition forms, we can always find an optimal bi-decomposition in the above three types. The reason is that a two-input function is always NP-equivalent to one of two-input AND, OR and XOR, and thus any bi-decomposition form can be treated as one of the above three bi-decomposition forms in terms of the input numbers of g_1 and g_2. Moreover, we can get g_1 and g_2 in an *OR-Decomposition* form by inverting g_1 and g_2 in an *AND-Decomposition* form of $\overline{f}$. Therefore, we must concentrate on *AND-Decomposition* and *XOR-Decomposition* forms. The way to efficiently find optimal *AND-Decomposition* and *XOR-Decomposition* forms will be shown in the rest of this paper.

3.2 Overview of How to Find *AND-Decomposition* and *XOR-Decomposition* forms

The overview of our methods for finding g_1 and g_2 in optimal *AND-Decomposition* and *XOR-Decomposition* forms for a given incompletely specified function f are as follows.

Step 1: generate an initial solution (g_1, g_2).

Step 2: improve the initial solution to produce an optimal solution based on a branch-and-bound algorithm.

Of course, Step 1 differs between *AND-Decomposition* and *XOR-Decomposition*, which we will explain in Section 4 and 5, respectively. However, the whole algorithm used in Step 2 is the same. It is a normal branch-and-bound algorithm as follows. For an initial solution, we treat the support sets of g_1 and g_2 as the same one as f. Then, at one step in our branch-and-bound algorithm, we consider the following three choices to improve an intermediate solution. For a variable in the support set of f that we have not tried before,

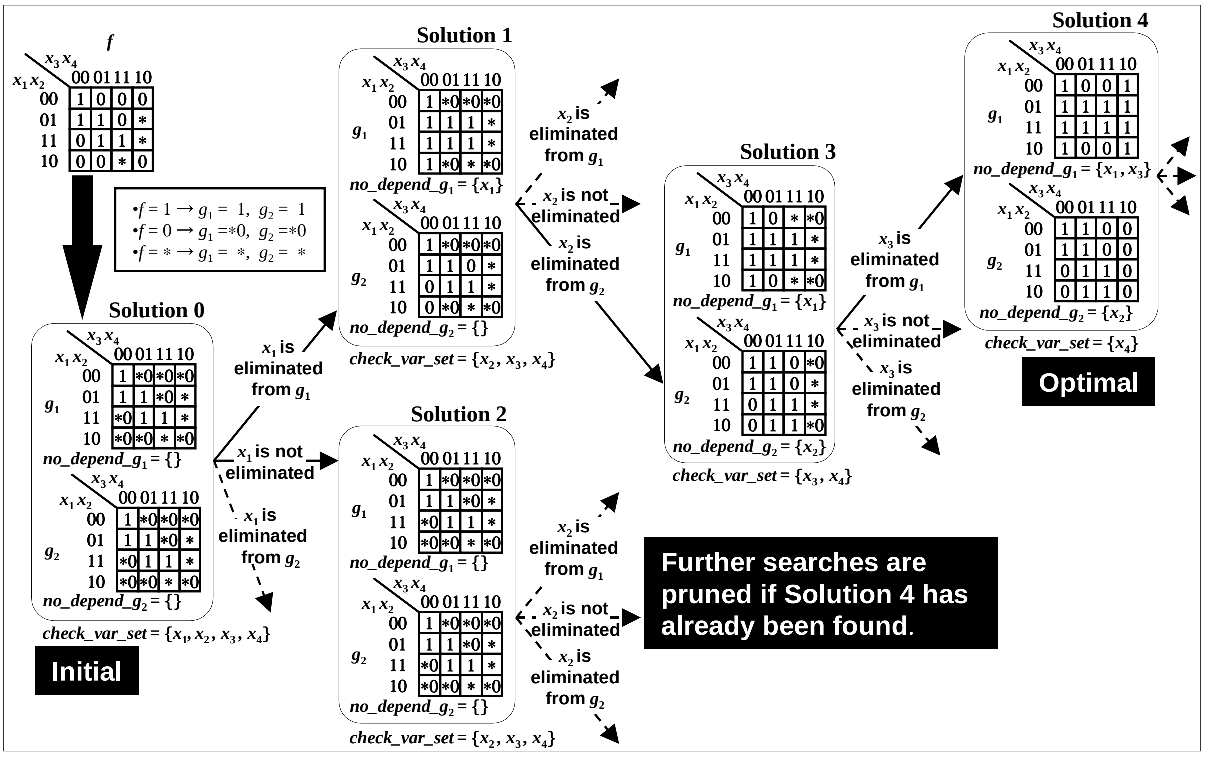

Fig. 2 A Search Tree for Finding an Optimal *AND-Decomposition*

Choice 1: try to eliminate the variable from the support set of g_1 to get a new solution.

Choice 2: try to eliminate the variable from the support set of g_2 to get a new solution.

Choice 3: do not eliminate the variable from the support set of either g_1 or g_2.

At least one of g_1 and g_2 depends on a variable in the support set of f. Therefore, we do not need to consider the case where a variable is eliminated from both the support sets of g_1 and g_2. In an optimal solution, there may be common variables in the support sets of g_1 and g_2, and thus we must check the case where a variable is not eliminated from the support set of either g_1 or g_2. This check can be done in Choice 3. In our branch-and-bound algorithm, we must keep the information concerning which variable has been eliminated from the support set of g_1 or g_2, and we use the following variables in this paper.

- *no_depend_*g_1 represents a set of variables that have already been eliminated from the support set of g_1 in our branch-and-bound process.
- *no_depend_*g_2 has the same meaning for g_2.

3.3 Our Branch-And-Bound Algorithm Example

Let us explain our branch-and-bound algorithm using an example for *AND-Decomposition*. A search tree is shown in Fig. 2. Suppose we want to find an optimal *AND-Decomposition* of function f: the ON-set is $(x_2 + \overline{x}_4) \cdot (x_1 \cdot x_4 + \overline{x}_1 \cdot \overline{x}_3)$, and the DC-set is $x_3 \cdot (x_2 \cdot \overline{x}_4 + x_1 \cdot \overline{x}_2 \cdot x_4)$. The truth table of f is shown at the top left-hand corner of Fig. 2. In the figure, $*$ means a usual don't care. "*check_var_set*" represents a set of variables for which we have not tried the above three choices. In other words, from an intermediate solution we must check further the variables in "*check_var_set*". Here, we do not explain how to generate "Solution 0" (initial solution) and the meaning of $*0$ in the figure. (They will be mentioned in Section 4.) We only want to explain the overview of our branch-and-bound algorithm which is common between *AND-Decomposition* and *XOR-Decomposition*. At "Solution 0", *no_depend_*g_1 and *no_depend_*g_2 are set to empty sets. From "Solution 0", we must check the above three choices for x_1, x_2, x_3 and x_4, and thus "*check_var_set*" is set to $\{x_1, x_2, x_3, x_4\}$. From "Solution 0", we try to eliminate x_1 from the support set of g_1, and we successfully get "Solution 1" in this case. From "Solution 1", we do not need to check the above three choices for x_1 any more. Therefore, we delete x_1 from "*check_var_set*" so that x_1 is not be eliminated from the support set of g_2 in a further search from "Solution 1".

Although there may be more than one optimal solution according to our definition in Section 2, we only find one of them. Therefore, a branch-and-bound algorithm is suitable for our purpose. This is because we successfully prune a search space as the normal branch-and-bound algorithm. For example, if "Solution 4" in Fig. 2 has already been found, we do not need to search an optimal solution from "Solution 2". This is because we know that we cannot find a better solution than "Solution 4" in the search space from "Solution 2" by only counting the numbers of variables in "*no_depend_g_1*", "*no_depend_g_2*" and "*check_var_set*". In the example, we successfully get "Solution 4" which is an optimal *AND-Decomposition*, where $g_1 = (x_2 + \overline{x}_4)$ and $g_2 = (x_1 \cdot x_4 + \overline{x}_1 \cdot \overline{x}_3)$.

In the overview, the following two points which are important in understanding our methods were not mentioned.

- How to generate an initial solution.
- How to eliminate a variable.

These two points are different for *AND-Decomposition* and *XOR-Decomposition*, and are explained in Section 4 and 5, respectively.

4. Finding an Optimal *AND-Decomposition* Form

In our method for finding an optimal *AND-Decomposition* form, g_1 and g_2 are treated as four-valued functions whose values are 0, 1, $*0$ or $*$. $*$ means a usual don't care. $*0$, which is introduced in this paper, has a special meaning as follows:

- One of g_1 or g_2 can be treated as usual don't care if the other is treated as 0.
- One of g_1 or g_2 must be 0 if the other is treated as 1.

In other words, $*0$ means that at least one of g_1 and g_2 must be 0. The introduction of $*0$ makes it possible to find an optimal solution based on a branch-and-bound algorithm.

In this paper, to explain how to transform four-valued and three-valued functions, we characterize a transformation of a function g as transformations of $ON(f)$, $OFF(f)$, $DC0(f)$ and $DC(f)$ which are defined as follows. In the definitions, g is a four-valued function and f is an incompletely specified function (three-valued) or a four-valued function.

- $ON(f)$ is a characteristic function that represents a set of minterms $\{a \mid f(a) = 1\}$.
- $OFF(f)$ is a characteristic function that represents a set of minterms $\{a \mid f(a) = 0\}$.
- $DC0(g)$ is a characteristic function that represents a set of minterms $\{a \mid g(a) = *0\}$.
- $DC(f)$ is a characteristic function that represents a set of minterms $\{a \mid f(a) = *\}$.

Note that all minterms must be included in one of the sets of minterms represented by $ON(f)$, $OFF(f)$, $DC0(f)$ and $DC(f)$. We sometimes represent a transformation of such as $ON(f)$ by adding or removing some minterms to the set of minterms represented by $ON(f)$. Moreover, we say only $ON(f)$ to represent the set of minterms represented by $ON(f)$ when the context is clear.

4.1 Generating an Initial Solution

In our method, initial g_1 is set to satisfy $ON(g_1) = ON(f)$, $DC0(g_1) = OFF(f)$, and $DC(g_1) = DC(f)$. Initial g_2 is set to the same as g_1. Clearly, this initial solution satisfies $f = g_1 \cdot g_2$. In the example in Fig. 2, the initial solution is shown as "Solution 0". The truth tables of initial g_1 and g_2 have $*$'s and $*0$'s. The main idea of our method is that by specifying these values to 1 or 0, we eliminate variables from the support set of g_1 or g_2.

4.2 Eliminating a Variable

The procedure "*ElimVarAnd*" in Fig. 3 eliminates x_i from the support set of g_1 in an intermediate solution if possible. The following conditions must be kept in *ElimVarAnd* so that we can utilize a branch-and-bound algorithm.

Condition 1: $f = g_1 \cdot g_2$.

Condition 2: g_1 does not depend on the variables in *no_depend_g_1*.

Condition 3: g_2 does not depend on the variables in *no_depend_g_2*.

To keep the above three conditions we must carefully perform the operations in *ElimVarAnd*. In *ElimVarAnd*, we use the following definitions of functions and operations. In the definitions, g is a four-valued function and h is a completely specified function (two-valued).

- h_{x_i} and $h_{\overline{x}_i}$ are the positive and negative cofactors of h with respect to x_i, respectively.
- *Smooth*(x_i, h) is $h_{x_i} + h_{\overline{x}_i}$.
- *SmoothSet*(var_set, h) is a function that is obtained by successively applying *Smooth*(x_i, h) to h for all x_i in *var_set*.

```
1   $ElimVarAnd(x_i,\ g_1,\ g_2,\ no\_depend\_g_2)\{$
2           /* to check if there is a possibility of eliminating $x_i$ from $g_1$ */
3       if $((Smooth(x_i, ON(g_1)) \cdot Smooth(x_i, OFF(g_1)))$ != the constant 0 function) return $FALSE$;
4           /* to calculate which minterms in $DC(g_1)$ and $DC0(g_1)$ must be moved */
5       $change1\_g_1 = (DC(g_1) + DC0(g_1)) \cdot Smooth(x_i, ON(g_1))$; /* needs to be changed to 1 */
6       $change0\_g_1 = (DC(g_1) + DC0(g_1)) \cdot Smooth(x_i, OFF(g_1))$; /* needs to be changed to 0 */
7       $EnlargeON(g_1, change1\_g_1)$;
8       $EnlargeOFF(g_1, change0\_g_1)$;
9           /* to calculate which minterms in $DC0(g_2)$ must be moved for Condition 1*/
10      $must\_change0\_g_2 = change1\_g_1 \cdot DC0(g_1)$; /* $DC0(g_1)$ is the same as $DC0(g_2)$ */
11          /* to enlarge $must\_change0\_g_2$ for Condition 3*/
12      $must\_change0\_g_2 = SmoothSet(no\_depend\_g_2, must\_change0\_g_2)$ ;
13          /* to check if $must\_change0\_g_2$ and $ON(g_2)$ have common minterms */
14      if $((ON(g_2) \cdot must\_change0\_g_2)$ != the constant 0 function) return $FALSE$;
15      $EnlargeOFF(g_2, must\_change0\_g_2)$;
16      $EnlargeDC(g_1, (DC0(g_1) \cdot OFF(g_2)))$;
17      $EnlargeDC(g_2, (DC0(g_2) \cdot OFF(g_1)))$;
18      return $TRUE$;
19  }
```

Fig. 3 *ElimVarAnd*

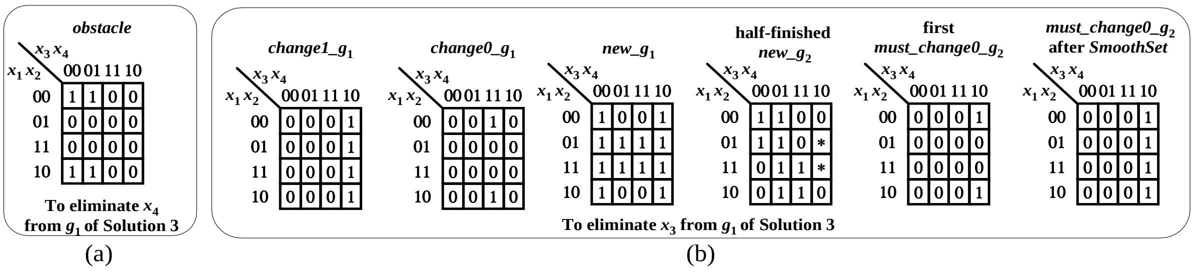
obstacle
x3 x4
x1 x2
00 01 11 10
To eliminate x4 from g1 of Solution 3
(a)
change1_g1
change0_g1
new_g1
half-finished new_g2
first must_change0_g2
must_change0_g2 after SmoothSet
To eliminate x3 from g1 of Solution 3
(b)

Fig. 4 Functions Explaining *ElimVarAnd*

- $EnlargeON(g, h)$ is an operation changing g so that $ON(g)$ is enlarged to $ON(g) + ON(h)$. In other words, this operation adds all minterms in $ON(h)$ to $ON(g)$ and at the same time removes the corresponding minterms from $OFF(g)$, $DC0(g)$ or $DC(g)$.
- $EnlargeOFF(g, h)$ is an operation changing g so that $OFF(g)$ is enlarged to $OFF(g) + ON(h)$.
- $EnlargeDC(g, h)$ is an operation changing g so that $DC(g)$ is enlarged to $DC(g) + ON(h)$.

ElimVarAnd consists of four steps as follows.

Step 1: transform g_1 so that x_1 is eliminated from the support set of g_1 if possible (at lines 2 to 8 in Fig. 3).

Step 2: transform g_2 to satisfy Condition 1 after the change of g_1 at Step 1 (at lines 9 to 10, and line 15 in Fig. 3).

Step 3: transform g_2 to satisfy Condition 3 if possible (at lines 11 to 15 in Fig. 3).

Step 4: transform g_1 and g_2 to enlarge their don't cares (at lines 16 to 17 in Fig. 3).

[Step 1]
The transformation of g_1 at Step 1 is characterized as follows: we move some minterms in the sets of minterms represented by $DC(g_1)$ and $DC0(g_1)$ to $ON(g_1)$ or $OFF(g_1)$ so that $ON(g_1)$ and $OFF(g_1)$ do not depend on x_i. We cannot do such a transformation if $(Smooth(x_i, ON(g_1)) \cdot Smooth(x_i, OFF(g_1)))$ is not the constant 0 function. This is checked at line 3 in Fig. 3, and $FALSE$ is returned if x_i cannot be eliminated. For example, we cannot eliminate x_4 from the support set of g_1 in "Solution 3" in Fig. 2. This is because $Smooth(x_4, ON(g_1)) \cdot Smooth(x_4, OFF(g_1))$ (shown as "*obstacle*" in Fig. 4 (a)) is not the constant 0 function. Indeed, we notice that x_4 cannot be eliminated from the support set of g_1 in "Solution 3" when we pay attention to the part of the truth table of g_1 in "Solution 3" which is masked by $ON($"*obstacle*"$)$.

$change1\_g_1$ and $change0\_g_1$ do not depend on the variables in $no\_depend\_g_1$, because g_1 does not depend on the variables in $no\_depend\_g_1$. Therefore, g_1 still does not depend on the variables in $no\_depend\_g_1$ after Step 1, that is, Condition 2 is satisfied after Step 1.

[Step 2]
After Step 1, g_1 is changed to a function that does not

depend on x_i. We need more operations in Step 2 to satisfy Condition 1. This is because $*0$ is not a usual don't care, and therefore, if a minterm in $DC0(g_1)$ is moved to $ON(g_1)$ by $EnlargeON(g_1, change1\_g_1)$, the corresponding minterm in $DC0(g_2)$ must be moved to $OFF(g_2)$.

[Step 3]
After Step 2, we get a solution where g_1 does not depend on x_i, and Conditions 1 and 2 are satisfied. However, Condition 3 is not always satisfied after Step 2 because there is no guarantee that $must\_change0\_g_2$ does not depend on the variables in $no\_depend\_g_2$. Thus, when we need to move a minterm m_i in $DC0(g_2)$ to $OFF(g_2)$, we must also move the minterms in $ON(SmoothSet(no\_depend\_g_2, ON(m_i)))$ to $OFF(g_2)$ so that g_2 does not depend on the variables in $no\_depend\_g_2$. Therefore, we must modify $must\_change0\_g_2$ at line 12 in Fig. 3. In some cases, the modified $must\_change0\_g_2$ and $ON(g_2)$ have common minterms. In such cases, we cannot do the operation at line 15 in Fig. 3 while keeping Condition 1. Therefore, in such cases *ElimVarAnd* returns *FALSE* (at line 14 in Fig. 3).

[Step 4]
If the minterms in $DC0(g_2)$ are moved to $OFF(g_2)$ during the above steps, the corresponding minterms in $DC0(g_1)$ can be moved to $DC(g_1)$. Therefore, we can enlarge the don't cares of g_1 by the operation at line 16 in Fig. 3. We can also enlarge the don't cares of g_2 by the operation at line 17 in Fig. 3.

4.3 An Example of *ElimVarAnd*

Here, we explain how to eliminate x_3 from the support set of g_1 in "Solution 3" to get "Solution 4" in Fig. 2. At first, "$change1\_g_1$" and "$change0\_g_1$" are calculated as shown in Fig. 4 (b). Then, we change g_1 by $EnlargeON(g_1, change1\_g_1)$ and $EnlargeOFF(g_1, change0\_g_1)$ to get new g_1 that does not depend on x_3. This is shown as "new_g_1" in Fig. 4 (b). Next, we calculate which minterms in $DC0(g_2)$ must be moved to $OFF(g_2)$ to satisfy Condition 1. We get the function which represents the minterms as "first $must\_change0\_g_2$" in Fig. 4 (b). If we adopt this for $must\_change0\_g_2$ for $EnlargeOFF(g_2, must\_change0\_g_2)$ at line 15 in Fig. 3, we get new g_2 as "half-finished new_g_2" in Fig. 4 (b). As previously mentioned, this new g_2 may not satisfy Condition 3. Indeed "half-finished new_g_2" in Fig. 4 (b) depends on x_2, which is a variable in $no\_depend\_g_2$. Therefore, we must modify "first $must\_change0\_g_2$" to "$must\_change0\_g_2$ after *SmoothSet*" in Fig. 4 (b) (at line 12 in Fig. 3). By adopting "$must\_change0\_g_2$ after *SmoothSet*" in Fig. 4 (b) for $EnlargeOFF(g_2, must\_change0\_g_2)$ at line 15 in Fig. 3, we successfully get the correct g_2 in "Solution 4" in Fig. 2.

In this way, we eliminate variables one by one from the support sets of g_1 and g_2 to find an optimal *AND-Decomposition*. Clearly our method can find one of the existing optimal solutions by elimination of any order of variables. The order only affects the execution time and which solution is found among the optimal solutions, if there is more than one optimal solution.

5. Finding an Optimal *XOR-Decomposition* Form

In our method for finding an optimal *XOR-Decomposition* form, unlike in the case of *AND-Decomposition*, we only use $*$ (usual don't care) and not $*0$. Therefore, g_1 and g_2 are treated as usual incompletely specified functions (three-valued) in this section. Although the main idea is almost the same as the case of *AND-Decomposition*, the way of generating an initial solution and eliminating a variable are apparently different from those in the case of *AND-Decomposition* because of the functional difference between AND and XOR. Therefore, we explain the above two points in this section.

5.1 Generating an Initial Solution

The initial g_1 is set to the same as f. The initial g_2 is set to the constant 0 function with the same don't cares as f. Clearly, this initial solution satisfies $f = g_1 \oplus g_2$.

5.2 Eliminating a Variable

The procedure "*ElimVarXor*" in Fig. 5 eliminates x_i from the support set of g_1 in an intermediate solution if possible. The following additional definitions of operations are used in this section.

- $Consensus(x_i, h)$ means $h_{x_i} \cdot h_{\overline{x}_i}$.
- $Reverse(g, h)$ means an operation to *reverse* the values of a part of the truth table of g as follows (*reverse* is an operation to change 1 to 0, 0 to 1, and $*$ to $*$).
  For each minterm m_i in $ON(h)$:
  - if m_i is in $ON(g)$, move m_i to $OFF(g)$.
  - if m_i is in $OFF(g)$, move m_i to $ON(g)$.

Of course, the following three conditions must be kept as the case of *AND-Decomposition*.

Condition 1: $f = g_1 \oplus g_2$.

Condition 2 and 3 are the same as the case of *AND-Decomposition*.

To satisfy Condition 1, we can only do a pair of operations: $Reverse(g_1, h)$ and $Reverse(g_2, h)$. This is because although we *reverse* the values of a part of the

```
1   $ElimVarXor(x_i,\ g_1,\ g_2,\ no\_depend\_g_1,\ no\_depend\_g_2)\{$
2           /* to calculate which minterms in $ON(g_1)$ and $OFF(g_1)$ must be moved */
3       $depending\_x_i\_g_1 = Smooth(x_i, ON(g_1)) \cdot Smooth(x_i, OFF(g_1))$;
4       $no\_depend\_All = no\_depend\_g_1 + no\_depend\_g_2$;
5           /* to enlarge $depending\_x_i\_g_1$ for Conditions 2 and 3*/
6       $depending\_x_i\_g_1 = SmoothSet(no\_depend\_All, depending\_x_i\_g_1)$;
7/* to check if $depending\_x_i\_g_1$ after $SmoothSet$ includes the part where $g_1$ must not be $reversed$ to eliminate $x_i$ */
8       if ( $((depending\_x_i\_g_1) \cdot Consensus(x_i, ON(g_1)))$ != the constant 0 function) return $FALSE$;
9       if ( $((depending\_x_i\_g_1) \cdot Consensus(x_i, OFF(g_1)))$ != the constant 0 function) return $FALSE$;
10      $must\_reverse = depending\_x_i\_g_1 \cdot \overline{x_i}$;
11      $Reverse(g_1, must\_reverse)$;
12      $Reverse(g_2, must\_reverse)$;
13      $change0 = Smooth(x_i, OFF(g_1)) \cdot DC(g_1)$;
14      $change1 = Smooth(x_i, ON(g_1)) \cdot DC(g_1)$;
15      $EnlargeOFF(g_1, change0)$;
16      $EnlargeON(g_1, change1)$;
17      return $TRUE$;
18  }
```

Fig. 5 *ElimVarXor*

truth table of g_1, $(g_1 \oplus g_2)$ does not change if the values of the same part of the truth table of g_2 are also *reversed*. Of course, we can freely change $*$'s of g_1 and g_2 to 1's or 0's. The main idea in *ElimVarXor* is that we restrict ourselves to only using the *reverse* operations for eliminating a variable.

First, we calculate which minterms in $ON(g_1)$ and $OFF(g_1)$ cause g_1 to depend on x_i. This calculation corresponds to $depending\_x_i\_g_1$ at line 3 in Fig. 5. x_i can be eliminated from the support set of g_1, if we do $Reverse(g_1, must\_reverse)$, where $must\_reverse$ satisfies the following two conditions, which we call "*Conditions for must_reverse*":

- $Smooth(x_i, must\_reverse) = depending\_x_i\_g_1$.
- $Consensus(x_i, must\_reverse)$ = the constant 0 function.

There are many candidates for $must\_reverse$ that satisfy the "*Conditions for must_reverse*". Among these candidates, we choose $depending\_x_i\_g_1 \cdot \overline{x_i}$ as $must\_reverse$ at line 10 in Fig. 5. Then, we change g_1 and g_2 by the operations at lines 11 and 12 in Fig. 5. To add to these operations, we must change a part of $DC(g_1)$ to $ON(g_1)$ or $OFF(g_1)$ so that g_1 does not depend on x_i. This is done at lines 13 to 16 in Fig. 5.

For Conditions 2 and 3, $must\_reverse$ must not depend on the variables in $no\_depend\_g_1$ and $no\_depend\_g_2$. Therefore, line 6 in Fig. 5 is needed.

Although we do the above operations, we cannot eliminate x_i from the support set of g_1 while keeping Conditions 1, 2 and 3. This check is done at lines 8 and 9 in Fig. 5.

There are many candidates for $must\_reverse$ that satisfy the "*Conditions for must_reverse*". Among these candidates, we choose $depending\_x_i\_g_1 \cdot \overline{x_i}$. If we choose another $must\_reverse$ among the candidates, we get another solution (g_1', g_2'). However, the variables which g_1' and g_2' depend on are the same for all candidates. Therefore, we can find one of the existing optimal solutions even though we choose $depending\_x_i\_g_1 \cdot \overline{x_i}$ among the many candidates for $must\_reverse$.

6. Experimental Results

We know the method presented here finds an "optimal" bi-decomposition form of a logic function. However, it is not certain that our method is really useful for logic synthesis. It is difficult to check whether it is good or not by itself because it might be a part of a logic synthesis system. Therefore, to compare only the decomposition power of our method with that of SIS 1.3 [10], we performed the following simple experiment on MCNC [11] benchmark circuits.

- First, we selected an output function with the largest number of variables and made a node whose function was an irredundant sum-of-products form of the selected function.
- We decomposed the nodes into 5-input LUTs by using "xl_imp -b" for SIS's script.
- For our results, we decomposed the node recursively to two-input nodes by finding optimal bi-decompositions. There are some functions that do not have optimal bi-decompositions. In other words, some functions do not have decomposition forms: $f(X) = \alpha(g_1(X^1), g_2(X^2))$ where the number of variables in X_1 and that in X_2 are both less than the number in X. We cannot decompose such functions by bi-decomposition, thus we only apply Shannon expansion for such functions.

Nodes with less than five inputs remained after the above decompositions. We merged such nodes to 5-

Table 1 Decomposition to 5-input LUTs

Circuits	SIS's results	Our results
(Nin)	LUT/lev/CPU	LUT/lev/CPU
alu2 (10)	33/6/12.68	12/5/1.83
b9 (14)	6/3/0.19	5/4/1.86
c8 (13)	7/3/0.23	4/3/0.66
cordic (23)	11/5/8.37	10/4/178.72
f51m (8)	12/3/0.83	3/3/0.21
frg1 (25)	30/7/10.41	17/4/1986.56
i8 (17)	12/8/0.75	11/6/26.39
lal (13)	5/3/0.25	4/3/1.75
pm1 (9)	3/3/0.13	3/2/0.05
sct (14)	4/4/0.22	7/3/1.78
t481 (16)	12/4/78.24	5/3/9.62
ttt2 (14)	7/4/0.44	4/3/2.9
x1 (25)	9/5/0.51	9/3/711.34
x3 (24)	11/5/0.83	14/5/161.59
x4 (15)	4/3/0.19	5/3/2.37
total	166/66/114.27	113/54/3087.63
ratio	1.00/1.00/1.00	0.68/0.82/27.02

input LUTs using a simple covering as shown in Fig. 1 (a).

Table 1 shows a comparison of our results and SIS's results. In Table 1, "LUT", "lev" and "CPU" show the number of LUTs, the network levels and the CPU run-time (sec.) on a Sun Ultra 2 2200, respectively. The ratios of our results to those of SIS are shown at the bottom. The numbers in the parentheses after the circuit names show the input numbers of the selected functions.

Although we only considered the aspect of decomposition in the experiment, we observed that good decompositions can be found by using optimal bi-decompositions. However, in some cases, our method took too much time. We think the main reason for this is the variable order which affects the pruning search space in our branch-and-bound algorithm as well as the efficiency of the BDD calculation used in our program. Therefore, we must develop a heuristic to find a better variable order in the future. We did not consider sharing common logics among several functions in the experiment, so we decomposed only one output function. We must extend the method to treat multi-output functions. We consider the following strategies for this purpose.

- When f is decomposed to $\alpha(g_1(X^1), g_2(X^2))$, we can check whether an existing function can be used as g_1 (or g_2) by the boolean resubstitution and the support minimization technique proposed in [5].

- After all decompositions, we can check whether a node can be replaced with another node by the method proposed in [12].

7. Conclusion and Future Work

We have presented a new efficient method for finding an "optimal" non-disjoint bi-decomposition form. Our method has the following properties.

- It uses a branch-and-bound algorithm that is suitable for finding an optimal solution.
- It eliminates variables one by one from the support set of g_1 and g_2 in an intermediate solution.
- It can find an optimal bi-decomposition by eliminating variables in any order.
- It can decompose incompletely specified functions.

In this paper, a bi-decomposition: $f(X) = \alpha(g_1(X^1), g_2(X^2))$ is "optimal" if the total number of variables in X^1 and X^2 is the smallest. Of course, there is no guarantee that the final decomposed networks are optimal even if we adopt "optimal" bi-decompositions at intermediate decompositions. However, we think our meaning of "optimal" is adequate if we want to decompose functions to LUTs.

There are some functions that do not have *non-trivial* bi-decompositions. These functions cannot be decomposed by bi-decomposition; Shannon expansion was only used for these functions in the experiment, which is not such a good strategy. Therefore, we plan to develop the proposed method by combining it with other decomposition methods. Moreover, we plan to extend our method to treat multi-output functions.

References

[1] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. CAD*, vol. CAD-6, pp. 1062–1081, Nov. 1987.

[2] G. Lee, D. Bang, and J. Saul, "Synthesis of LUT-type FPGAs using AND/OR/EXOR representations," in *Proc. SASIMI*, pp. 74–77, Nov. 1996.

[3] Y. Matsunaga, "An Attempt to Factor Logic Functions Using Exclusive-Or Decomposition," in *Proc. SASIMI*, pp. 78–83, Nov. 1996.

[4] R. L. Ashenhurst, "The Decomposition of Switching Functions," in *Proceedings of an International Symposium on the Theory of Switching*, pp. 74–116, Apr. 1957.

[5] H. Sawada, T. Suyama, and A. Nagoya, "Logic Synthesis for Look-up Table Based FPGAs Using Functional Decomposition and Support Minimization," in *Proc. ICCAD*, pp. 353–358, Nov. 1995.

[6] J. P. Roth and R. M. Karp, "Minimization Over Boolean Graphs," *IBM journal*, pp. 227–238, Apr. 1962.

[7] T. Sasao and J. T. Butler, "On Bi-Decompositions of Logic Functions," in *Notes of International Workshop on Logic Synthesis (IWLS'97)*, May 1997.

[8] D. Bochmann, F. Dresig, and B. Steinbach, "A new decomposition method for multilevel circuit design," in *Proc. EDAC*, pp. 374–377, Feb. 1991.

[9] B. Steinbach and A. Wereszczynski, "Synthesis of Multi-Level Circuits Using EXOR-Gates," in *Proc. of Reed-Muller'95*, pp. 161–168, Aug. 1995.

[10] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Univ. of California, Berkeley, May 1992.

[11] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," *MCNC*, Jan. 1991.

[12] S. Yamashita, H. Sawada, and A. Nagoya, "A New Method to Express Functional Permissibilities for LUT based FPGAs and Its Applications," in *Proc. ICCAD*, pp. 254–261, Nov. 1996.

Shigeru Yamashita received his B. E. and M. E. degrees in information science from Kyoto University, Kyoto, Japan, in 1993 and 1995, respectively. In 1995, he joined NTT Communication Science Laboratories, where he has been engaged in research of computer aided design of digital systems and computer architecture. He is a member of the Information Processing Society of Japan.

Hiroshi Sawada was born in Osaka, Japan, on October 31, 1968. He received the B. E. and M. E. degrees in information science from Kyoto University, Kyoto, Japan, in 1991 and 1993, respectively. In 1993, he joined NTT Communication Science Laboratories, where he has been engaged in research of computer aided design of digital systems and computer architecture. He is a member of the Information Processing Society of Japan.

Akira Nagoya received his B. E. and M. E. degrees in electronic engineering from Kyoto University in 1978 and 1980, respectively. He joined the NTT Electrical Communication Laboratories in 1980 where he is now a Research Group Leader at the Communication Science Laboratories. From 1980 to 1987, he was engaged in research and development of mainframe CPU architecture. Since 1989, he has been engaged in research of CAD system for ASIC design. From 1990 to 1991, he was with the Department of Computer Science, University of Illinois at Urbana-Champaign, as a visiting scholar. His areas of research interest are logic synthesis, high-level synthesis, and computer architecture design. He is a member of the Information Processing Society of Japan. He received the Okochi Memorial Technology Prize in 1992.