Unified Functional Decomposition via Encoding for FPGA Technology Mapping

Jie-Hong Jiang, Jing-Yang Jou, *Member, IEEE*, and Juinn-Dar Huang

***Abstract*—Functional decomposition has recently been adopted for look-up tabel (LUT)-based field-programmable gate array (FPGA) technology mapping with good results. In this paper we propose a novel method to unify functional single-output and multiple-output decomposition. We first address a compatible class encoding algorithm to minimize the number of compatible classes in the image function. After applying the encoding algorithm, we can therefore improve the decomposability in the subsequent decomposition of the image function. The above encoding algorithm is then extended to encode multiple-output functions through the construction of a hyperfunction. Common subexpressions among these multiple-output functions can be extracted during the decomposition of the hyperfunction. Consequently, we can handle multiple-output decomposition in the same manner as single-output decomposition. Experimental results show that our algorithms are promising.**

***Index Terms*—Compatible class encoding, FPGA, functional decomposition, technology mapping.**

I. Introduction

FIELD-PROGRAMMABLE gate arrays (FPGAs) can provide programmability for users to implement their own logic. Because of the short turnaround time, FPGAs become increasingly popular in rapid system prototyping. look-up table (LUT)-based architecture is a prevalent one among many FPGA architectures. LUT-based FPGAs consist of an array of LUTs, each of which can implement any Boolean function with up to k (four or five in general) inputs, that is, k-feasible. A Boolean network can be directly realized by a one-to-one mapping between nodes and LUTs if every node in the network has up to k supports.

Functional decomposition [1]–[3] is a pivotal decomposition technique for LUT-based FPGA logic synthesis. Three interesting problems in functional decomposition should be noticed.

1) How to select bound set variables?
2) How to encode compatible classes?
3) How to extract adequate subexpressions among multiple-output functions?

Algorithms proposed in [4] and [5] provide solutions to choose good bound set variables. On the other hand, approaches suggested in [6]–[11] deal with the second problem. These encoding algorithms can be classified into three classes according to their objectives. The first kind of encoding algorithm, such as [6], simplifies the image function. The second kind, such as [7] and [8], makes multiple-output functions able to share some decomposition functions. The third kind, such as [9]–[11], minimizes the supports of the decomposition functions. As Problem 3 is considered, approaches proposed in [7], [8], and [12] extract common subexpressions among multiple-output functions. Extracting common decomposition functions via compatible class encoding was suggested in [7] and [8]. Sawada *et al.* in [12] tried to resubstitute decomposition functions into other functions to reduce their supports. In other words, common subexpressions were extracted by resubstitution.

In this paper, we adopt the variable partitioning algorithm proposed in [5], which takes advantage of binary decision diagrams (BDDs) [7], [13] to conduct functional decomposition to solve Problem 1. We will thus focus on solving Problems 2 and 3 in this paper. A new encoding algorithm is proposed to simplify the image function. Instead of reducing the number of cubes or literals in the image function as suggested in [6], our encoding algorithm aims at reducing the compatible class count. Thus a more precise estimation of LUT-costs can be derived during functional decomposition in FPGA technology mapping. To deal with Problem 3, we transform multiple-output functions into a single-output function by introducing the hyperfunction approach. Consequently, multiple-output decomposition can be reduced to an equivalent single-output decomposition. It unifies the solutions of single-output and multiple-output decomposition. Besides, the extracted common subexpression can be large.

The rest of this paper is organized as follows. Section II introduces preliminaries. In Section III, the compatible class encoding algorithm is proposed. Hyperfunction decomposition is then discussed in Section IV. After experimental results are shown in Section V, concluding remarks will be given in Section VI.

II. Preliminaries

Let $B = \{0, 1\}$. A single-output function f with n input variables $b_0, \ldots$ and b_{n-1} is denoted as $f\colon B^n \rightarrow B$. A function $f(b_0, \ldots, b_{n-1})$, is **decomposable** if it can be represented by another function $g(\overline{\alpha}\ (b_0, \ldots, b_{i-1}),\ b_j, \ldots, b_{n-1}) = g(\alpha_0(b_0, \ldots, b_{i-1}), \ldots, \alpha_{t-1}(b_0, \ldots, b_{i-1}), b_j, \ldots, b_{n-1})$, where $0 < t < j \leq i$. The decomposition is disjoint if $j = i$. When $j = i, \{b_0, \ldots, b_{i-1}\}$ is called the **bound ($\boldsymbol{\lambda}$) set** and $\{b_j, \ldots, b_{n-1}\}$ is called the **free ($\boldsymbol{\mu}$) set**. In this paper, only disjoint decomposition is considered, that is, the bound set and free set are disjoint. In this article, the g function is referred to

Manuscript received July 14, 1998. This work was supported in part by the National Science Council of Taiwan ROC under Grant NSC89-2215-E-009-009.

J.-H. Jiang was with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan. He is now with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 USA.

J.-Y. Jou is with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan.

J.-D. Huang is with the UniChip Corporation, Hsinchu, Taiwan.

Publisher Item Identifier S 1063-8210(01)00702-8.

as the **image function** and the $\overline{\alpha}$ functions are referred to as the **decomposition functions**.

Definition 1: Let X and Y be two sets of binary variables, $X \cap Y = \emptyset$. Given a completely specified function f: $B^{|X|} \times B^{|Y|} \rightarrow B$, with X being the λ set and Y being the μ set. We say that $x_1, x_2 \in B^{|X|}$ are **compatible** with respect to f, denoted as $x_1 \sim x_2$, if $\forall y \in B^{|Y|}$, (x_1, y) and $(x_2, y) \in B^{|X|} \times B^{|Y|}$ such that $f(x_1, y) = f(x_2, y)$. All mutually compatible elements form a **compatible class**.

Theorem 1: $\forall (x, y) \in B^{|X|} \times B^{|Y|}$, f: $B^{|X|} \times B^{|Y|} \rightarrow B$, $\overline{\alpha} : B^{|X|} \rightarrow W, g : W \times B^{|Y|} \rightarrow B$,

$$f(x, y) = g\big(\overline{\alpha}\ (x), y\big) \tag{1}$$

holds if and only if

$$\forall x_1, x_2 \in B^{|X|}, \quad \overline{\alpha}\ (x_1) = \overline{\alpha}\ (x_2) \Rightarrow x_1 \sim x_2. \tag{2}$$

$\overline{\alpha}$ is a function with binary inputs and a symbolic output. The number of the admissible values in W, $|W|$, must be no less than the number of compatible classes. To implement $\overline{\alpha}$ by binary logic, at least $t = \lceil \log_2 |W| \rceil$ binary functions, $\alpha_0, \ldots, \alpha_{t-1}$, are required to encode $\overline{\alpha}$. When $t = \lceil \log_2 |W| \rceil$, we say that this encoding is **rigid**. Otherwise when $t > \lceil \log_2 |W| \rceil$, the encoding is **pliable**. Equation (1) can be rewritten as $f(x, y) = g(\alpha_0(x), \ldots, \alpha_{t-1}(x), y)$. For a single-output function, if each compatible class is encoded by just one code, then this encoding is **strict** and (2) can be redefined as

$$\forall x_1, x_2 \in B^{|X|}, \quad \overline{\alpha}\ (x_1) = \overline{\alpha}\ (x_2) \Leftrightarrow x_1 \sim x_2.$$

In contrast, if there exists any compatible class encoded with more than one code, then the encoding is **nonstrict**.

III. Compatible Class Minimization

Two important factors affect the decomposition quality: one is variable partitioning and the other is compatible class encoding. We solve the variable partitioning problem by using the algorithm proposed in [5]. Thereby we focus on compatible class encoding to reduce the number of compatible classes in the subsequent decomposition of the image function. Before the discussion of the encoding technique in Section III-B, don't care assignment, which is used in the encoding, is introduced first in Section III-A.

A. Don't Care Assignment

Sawada *et al.* in [12] used the don't care assignment to minimize the supports of an incompletely specified function. In contrast, we formulate the don't care assignment problem as graph clique-partitioning in order to reduce the number of compatible classes instead of supports.

We record the compatible relationship among λ set vertices by using the compatibility graph. Each λ set vertex corresponds to a vertex in the graph. A pair of vertices are connected by an edge if and only if these two vertices are compatible under certain don't care assignment. After constructing the graph, we want to find the least number of cliques such that each vertex is exactly covered by one clique. Since the clique partitioning problem is NP-complete, we adopt the heuristics in [14] to have a polynomial time approximation. After the graph is covered by cliques, the number of cliques equals the number of compatible classes after don't care assignment.

B. Compatible Class Encoding

After λ set selection and don't care assignment, compatible classes are fully determined. The next step is to encode these compatible classes.

Murgai *et al.* in [6] assumed that the fewer cubes or literals in the image function, the better decomposition quality could be obtained. As a result, the compatible class encoding problem was modeled as the symbolic-input encoding problem to minimize the number of cubes or literals of the image function. However, literal or cube counts may not be a good cost function for LUT-based FPGA synthesis. In this paper, we formulate the objective of the encoding problem as minimizing the number of compatible classes in the subsequent decomposition of the image function. The new cost function has better meaning for LUT architecture. (In order to exploit more don't care set, this paper takes the strict encoding policy.)

Example 1 illustrates why an encoding is relevant to the number of compatible classes in the decomposition of the image function.

Example 1: Assume that the targeted LUT can implement any function with up to four inputs. In addition, suppose the function f under decomposition is in Fig. 1(a) with $\{a, b, c\}$ as the λ set variables.

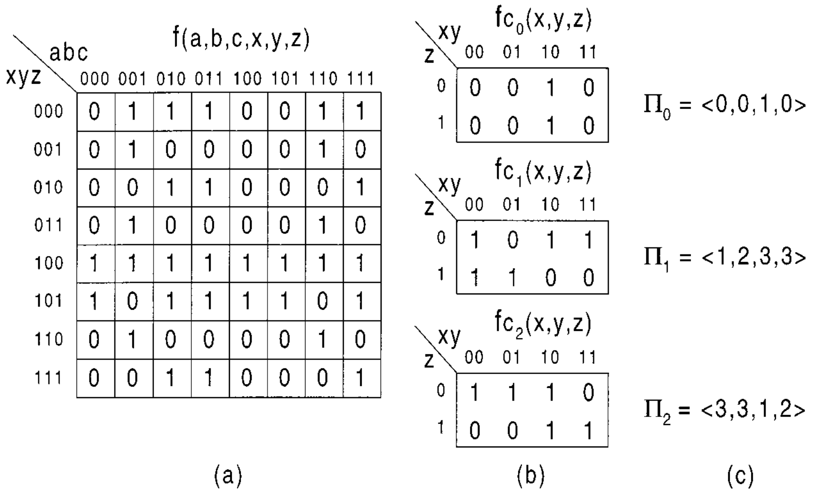
According to the decomposition chart in Fig. 1(a), there are three compatible classes with functions as shown in Fig. 1(b). For rigid encoding, two α-functions, $\alpha_0(a, b, c)$ and $\alpha_1(a, b, c)$, are needed to encode three compatible classes. Suppose we further choose α_0, x, and y as the λ set variables in the decomposition of $g(\alpha_0, \alpha_1, x, y, z)$. Examining the two encoding cases in Fig. 2, we can see that the encoding may affect the number of compatible classes in the decomposition of the g-function. Note that "$-$" represents don't care.

From Example 1, we can conclude the following theorems.

Theorem 2: If all of the α-functions are selected together in the λ set or μ set in the subsequent decomposition of the image function, the encoding does not affect the number of compatible classes in the subsequent decomposition of the image function.

Theorem 3: After the λ set variables used in the decomposition of an image function have been selected, to reduce the number of compatible classes of the image function, we only have to determine which compatible class functions should belong to the same column or the same row in the encoding chart. Exact codes of these columns and rows do not influence the number of compatible classes of the image function.

According to Theorem 3, in order to reduce the number of compatible classes when decomposing an image function, we must reduce the column patterns in its decomposition chart by assigning suitable compatible classes into the same column or the same row in the encoding chart. Because the encoding problem is too difficult to be solved exactly, we propose a heuristic algorithm, which considers the assignments of columns and rows separately. The encoding flow is shown in Fig. 3. We detail the encoding algorithm with Example 2.



f(a,b,c,x,y,z)

xyz \ abc	000	001	010	011	100	101	110	111
000	0	1	1	1	0	0	1	1
001	0	1	0	0	0	0	1	0
010	0	0	1	1	0	0	0	1
011	0	1	0	0	0	0	1	0
100	1	1	1	1	1	1	1	1
101	1	0	1	1	1	1	0	1
110	0	1	0	0	0	0	1	0
111	0	0	1	1	0	0	0	1

(a)

$fc_0(x,y,z)$

z \ xy	00	01	10	11
0	0	0	1	0
1	0	0	1	0

$fc_1(x,y,z)$

z \ xy	00	01	10	11
0	1	0	1	1
1	1	1	0	0

$fc_2(x,y,z)$

z \ xy	00	01	10	11
0	1	1	1	0
1	0	0	1	1

(b)

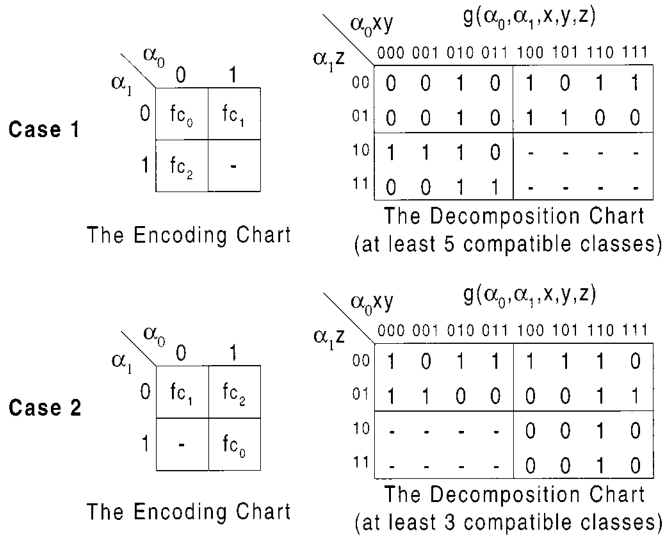
$\Pi_0 = \langle 0,0,1,0 \rangle$

$\Pi_1 = \langle 1,2,3,3 \rangle$

$\Pi_2 = \langle 3,3,1,2 \rangle$

(c)

Fig. 1. (a) Decomposition chart of f, (b) compatible class functions, and (c) symbolic representation of column patterns. (Symbols 0, 1, 2, and 3 represent column patterns $[00]^T$, $[11]^T$, $[01]^T$, and $[10]^T$, respectively.)



Case 1

α_1 \ α_0	0	1
0	fc_0	fc_1
1	fc_2	-

The Encoding Chart

$g(\alpha_0,\alpha_1,x,y,z)$

$\alpha_1 z$ \ $\alpha_0 xy$	000	001	010	011	100	101	110	111
00	0	0	1	0	1	0	1	1
01	0	0	1	0	1	1	0	0
10	1	1	1	0	-	-	-	-
11	0	0	1	1	-	-	-	-

The Decomposition Chart (at least 5 compatible classes)

Case 2

α_1 \ α_0	0	1
0	fc_1	fc_2
1	-	fc_0

The Encoding Chart

$g(\alpha_0,\alpha_1,x,y,z)$

$\alpha_1 z$ \ $\alpha_0 xy$	000	001	010	011	100	101	110	111
00	1	0	1	1	1	1	1	0
01	1	1	0	0	0	0	1	1
10	-	-	-	-	0	0	1	0
11	-	-	-	-	0	0	1	0

The Decomposition Chart (at least 3 compatible classes)

Fig. 2. Encoding chart and decomposition chart.

In the rest of this paper, we will use symbolic notations (decimal numbers) to represent column patterns. Some terminologies and notations are defined as follows.

Definition 2: A **partition Π**, $\langle s_0, \ldots, s_{n-1}\rangle$, is a symbolic notation of n column patterns. Element s_i equals s_j if and only if the ith column pattern equals the jth column pattern.

For example in Fig. 1, partitions Π_0, Π_1, and Π_2 in (c) are symbolic notations of column patterns in the charts of fc_0, fc_1 and fc_2 in (b), respectively. A **conjunctive partition Πc** (**disjunctive partition Πd**) of a set of partitions is a new partition which is a symbolic notation of column patterns formed by stacking these partitions vertically in the same column (horizontally in the same row) of the encoding chart. The **multiplicity** of a partition is the number of different symbols in this partition.

Example 2: Assume we have ten compatible class functions (fc_0, ... and fc_9) as the input of the encoding algorithm, and, after variable partitioning in Step 3 of Fig. 3, their partitions become $\Pi_0 = \langle 0,1,2,3\rangle$, $\Pi_1 = \langle 0,2,1,3\rangle$, $\Pi_2 = \langle 3,0,1,3\rangle$, $\Pi_3 = \langle 2,1,0,1\rangle$, $\Pi_4 = \langle 0,1,3,1\rangle$, $\Pi_5 = \langle 0,1,0,2\rangle$, $\Pi_6 = \langle 1,0,0,0\rangle$, $\Pi_7 = \langle 1,1,2,1\rangle$, $\Pi_8 = \langle 1,2,1,2\rangle$, and $\Pi_9 = \langle 3,2,1,0\rangle$, respectively. Suppose we want to place these partitions (or these compatible class functions) in an encoding chart with $\#R = 4$ and $\#C = 4$.

Step 5 in Fig. 3: *Evaluate which compatible classes should be bounded in the same column of the encoding chart.*

To decide which partitions should be placed in the same column in an encoding chart, we form conjunctive partitions on them. Smaller multiplicity of a conjunctive partition of k partitions is preferred, especially when $k \geq \#R$ (since $\#R$ of these partitions can be stacked in one column in an encoding chart).

We represent position i in a partition as p_i for convenience. Since the contents of p_1 and p_3 in Π_4 are the same, we say that positions with the same content of Π_4 is $p_1 p_3$. For the above ten partitions, Fig. 4(a) records the information. Positions with the same content such as $p_0 p_3$, we denote it as Psc_{03}. $|Psc_{03}| = 2$ because there are two positions, p_0 and p_3, in Psc_{03}. As there are two partitions Π_2 and Π_7 having Psc_{03}, we say #Partitions$(Psc_{03}) = 2$ and Partitions$(Psc_{03}) = \{\Pi_2, \Pi_7\}$. The conjunctive partition of Π_2 and Π_7 will thus have the same content in p_0 and p_3. For all Psc's in Fig. 4(a) ($Psc_{03}, Psc_{13}, Psc_{02}, Psc_{123}$, or Psc_{013}) with #Partitions$(Psc) \geq 2$, we list them in Fig. 4(b). We then build a *column-graph* $Gc(Vc, Uc, Ec)$ as depicted in Fig. 5, which is a bipartite graph. For each partition, there is a corresponding vertex in Vc; for each Psc with #Partitions$(Psc) \geq 2$, there are $\lceil(\#\text{Partitions}(Psc) - 1)/\#R\rceil$ corresponding vertices, u_{Psc}'s, in Uc. (It is because Partitions(Psc) may be collected in more than one column set when $\lceil(\#\text{Partitions}(Psc)-1)/\#R\rceil$ is not less than two.) A vertex in Uc corresponding to a Psc has #Partitions(Psc) edges connecting this vertex with vertices in Vc which are corresponding to Partitions(Psc). The weight of an edge connecting to u_{Psc} equals ($|Psc|+$ the number of edges connecting to u_{Psc}). We then find a *b-matching* [15], Mc, of maximum weight for Gc. After b-matching, each vertex in Vc must be connected by at most one edge; each vertex in Uc must be connected by at most $\#R$ edges. Finally, the corresponding partitions of vertices $\in Vc$ connected with the same

```
Algorithm: Encoding
Input:   Compatible Class Functions fc's
Output:  g-function with Its $\lambda$ Set $\lambda_g$, Encoding Result E
begin
1    {g',E'} ← Encode fc's at random;
2    if ( g' is $\kappa$–feasible )   return {g',∅ ,E'};
3    {$\lambda_g$,#compatible_class} ← Variable_Partitioning (g');
     /*From g', $\lambda_g$ and fc's, we can derive the # of rows, #R, the # of
     columns, #C, in the encoding chart and the partitions of fc's, $\Pi_{fc}$'s.*/
4    if ( #R==1 or #C==1 ) return {g',$\lambda_g$,E'};  /*According to Theorem 2*/
     /*Each $\Pi_{fc}$ occupies a distinct row set and a distinct column set initially.*/
5    CSet's ← Combine_Column_Sets ($\Pi_{fc}$'s);
6    while ( |RSet's|> #R or |CSet's|> #C )
7        {RSet's,CSet's} ← Combine_Row_Sets (RSet's,CSet's);
8    if ( random encoding has better results )  return {g',$\lambda_g$,E'};
9    {g,E} ← Encoding according to RSet's and CSet's;
10   return {g,$\lambda_g$,E};
end
```

Fig. 3. The encoding procedure.

vertex in Uc are grouped in the same column set. According to the matching result in Fig. 5, we have six column sets, $\{\Pi_3, \Pi_4, \Pi_6, \Pi_8\}$, $\{\Pi_2,\Pi_7\}$, $\{\Pi_0\}$, $\{\Pi_1\}$, $\{\Pi_5\}$, and $\{\Pi_9\}$. After applying the b-matching, we can reduce the multiplicity of the Πc of partitions in the same column set.

Step 7 in Fig. 3: *Evaluate which compatible classes should be bounded in the same row and in adjacent columns of the encoding chart.*

We like to bound a set of partitions in the same row if the multiplicity of their disjunctive partition is small. Because the row is stacked with other rows eventually, the multiplicity of the conjunctive partition of these stacked rows could be smaller.

Assume each partition initially occupies a separate row set. We calculate the benefits of merging pairs of row sets. For Π_i and Π_j in different column sets, we calculate the benefit of merging them as $(\sigma \times Br_{ij} + \tau \times Bc_{ij})$. When Π_i and Π_j are in the same column set, we don't want to put them into different columns. Therefore the benefit is calculated as ($\sigma \times Br_{ij} + \tau \times Bc_{ij}$—the weight of the edge connecting the vertex corresponding to Π_i in Vc of Gc). The calculations of σ, Br_{ij}, τ, and Bc_{ij} are as follows:

$$\begin{aligned}
Br_{ij} &= n - (n_{ij} - n_i) - (n_{ij} - n_j)\\
Bc_{ij} &= \sum_{\text{every symbol } S \text{ in all } \Pi\text{'s}} ((\#\text{ of } S \text{ in } \Pi_i \text{ and } \Pi_j) - k)^2\\
\sigma &= (\#\text{ of row sets so far}) - \#R\\
&\qquad (\text{if } \sigma < 0 \Rightarrow \sigma = 0)\\
\tau &= (\#\text{ of column sets so far}) - \#C\\
&\qquad (\text{if } \tau < 0 \Rightarrow \tau = 0)\\
k &= m/n
\end{aligned}$$

where m is the # of positions in the Πd of Π_i and Π_j; n, n_{ij}, n_i, and n_j are the total numbers of distinct symbols in all partitions, in Πd of Π_i and Π_j, in Π_i and in Π_j, respectively.

According to the calculated benefits, we construct a *row-graph* $Gr(Vr, Er)$. Each partition Π_i has a corresponding vertex $v_i \in Vr$; each pair of vertices (v_i, v_j) is connected by an undirected edge whose weight is the benefit of merging Π_i and Π_j in the same row set. We then find the *maximum-cardinality matching* [15], Mr, of Gr. For each edge $\in Mr$, the corresponding partitions of its two end vertices are hopefully to be combined together in a row set. We combine these pairs of partitions iteratively with benefits from high to low until the number of current row sets is not greater than $\#R$ or all edges $\in Mr$ have been selected. In this example, $\{\Pi_7, \Pi_8\}, \{\Pi_5, \Pi_6\}, \{\Pi_2, \Pi_4\}, \{\Pi_0, \Pi_9\}$, and $\{\Pi_1, \Pi_3\}$ are therefore selected in succession. According to column sets derived in Step 5, we stack these pairs of partitions properly. If there are some conflicts between the results of Step 5 and those of Step 7, we assume that the decisions of Step 7 have higher priority than those of Step 5. So far we have five row sets $\{\Pi_7, \Pi_8\}$, $\{\Pi_5, \Pi_6\}$, $\{\Pi_2, \Pi_4\}$, $\{\Pi_0, \Pi_9\}$, $\{\Pi_1, \Pi_3\}$, and four column sets $\{\Pi_3, \Pi_4, \Pi_6, \Pi_8\}$, $\{\Pi_1, \Pi_2, \Pi_5, \Pi_7\}$,

Partition	Positions with the same content
Π_2	p0p3
Π_3	p1p3
Π_4	p1p3
Π_5	p0p2
Π_6	p1p2p3
Π_7	p0p1p3
Π_8	p0p2, p1p3

(a)

Partition	Positions with the same content
Πc of $\{\Pi_2,\Pi_7\}$	p0p3
Πc of $\{\Pi_3,\Pi_4,\Pi_6,\Pi_7,\Pi_8\}$	p1p3
Πc of $\{\Pi_5,\Pi_8\}$	p0p2

(b)

Fig. 4. Partitions and positions with the same contents.

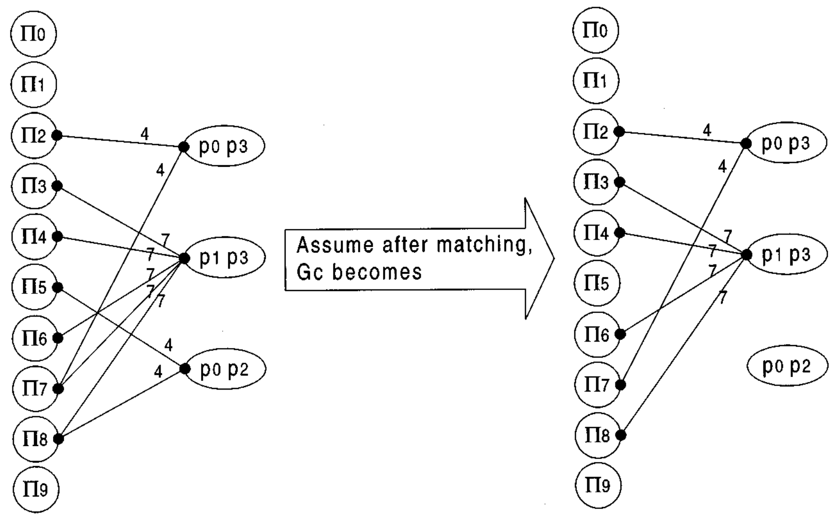


Fig. 5. Graph formulation of column-set combination.

$\{\Pi_0, \Pi_9\}$. We have row-column relations as illustrated in Fig. 6(a).

Step 7 must be repeated until the number of row sets $\leq \#R$ and the number of column sets $\leq \#C$. In this example so far, since there are five row sets $(>\#R)$, we thus iterate Step 7 to calculate the benefits of combining pairs of row sets. For each row set, we represent it by the disjunctive partition of the partitions in this row set as shown in Fig. 6(b). For instance, row set $\{\Pi_7, \Pi_8\}$ is represented as Πd_{78}. After constructing the new $Gr'(Vr', Er')$ and finding the Mr' for it, we combine the pair of row sets with maximum benefits $(\{\Pi_1, \Pi_3\}, \{\Pi_0, \Pi_9\})$ into a row set $\{\Pi_1, \Pi_3, \Pi_0, \Pi_9\}$. Since the number of row sets becomes four $(<\#R)$, we don't have to combine row sets further. Finally as shown in Fig. 7(a), we have four row sets and four column sets. According to Theorem 3, we know that the codes of columns and rows do not affect the number of compatible classes in the decomposition of the image function. So we can encode these compatible classes as that indicated in Fig. 7(b). After encoding, we can derive the real image function. In the decomposition of this image function, we will select α_0, α_1 and the other two bound set variables in the previous variable partitioning (derived in Step 3) as the bound set variables. Consequently, we will have four compatible classes in the subsequent decomposition of the image function.

Because we look one step ahead to collect information in the subsequent decomposition of the image function, we have more chances to reduce the number of compatible classes. Note that even when the number of bits needed to encode compatible classes may not be reduced in some cases, reducing the number of compatible classes produces larger don't care set in the image function and improves the decomposability.

IV. Hyperfunction Decomposition

To solve multiple-output decomposition in the same manner as single-output decomposition, we propose a solution, which is called the hyperfunction decomposition.

Definition 3: A set of distinct Boolean functions $\{f_0, \ldots, f_{n-1}\}$, called **ingredients**, can be combined together to form a single-output **hyperfunction** by using additional $i = \lceil \log_2 n \rceil$ binary bits $\eta_0, \ldots, \eta_{i-1}$, called **pseudo primary inputs**, to encode these ingredients. The supports of a hyperfunction include pseudo primary inputs and the union of supports of its ingredients.

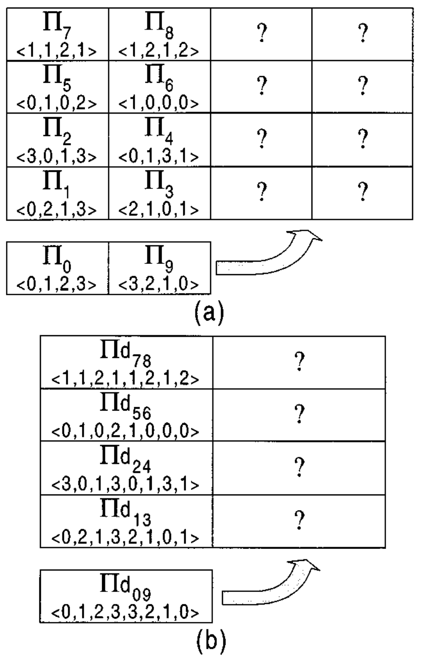
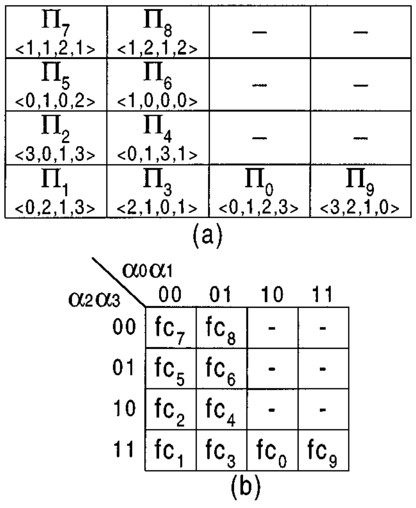


Fig. 6. Row-column relation.

A hyperfunction transforms multiple-output functions into a single-output function. It can perform the functionality of any individual ingredient by assigning the ingredient's corresponding code to pseudo primary inputs. In Section IV-A, we discuss how to encode ingredients to construct a hyperfunction with better decomposability.

A. Encoding of Ingredients

Actually, a hyperfunction can be considered as an image function, and its ingredients can be viewed as compatible class func-



Π_7 <1,1,2,1>	Π_8 <1,2,1,2>	–	–
Π_5 <0,1,0,2>	Π_6 <1,0,0,0>	–	–
Π_2 <3,0,1,3>	Π_4 <0,1,3,1>	–	–
Π_1 <0,2,1,3>	Π_3 <2,1,0,1>	Π_0 <0,1,2,3>	Π_9 <3,2,1,0>

(a)

$\alpha_2\alpha_3 \backslash \alpha_0\alpha_1$	00	01	10	11
00	fc_7	fc_8	-	-
01	fc_5	fc_6	-	-
10	fc_2	fc_4	-	-
11	fc_1	fc_3	fc_0	fc_9

(b)

Fig. 7. (a) Final row-column relation and (b) final encoding.

tions. Consequently, Theorems 2 and 3 can be extended as follows.

Theorem 4: If all of the pseudo primary inputs are selected together in the λ set or μ set in the decomposition of a hyperfunction, encoding does not affect the number of compatible classes in the decomposition of the hyperfunction.

Theorem 5: After the λ set variables used in the decomposition of a hyperfunction have been selected, to reduce the number of compatible classes of the hyperfunction, we only have to determine which ingredients should belong to the same column or the same row in the encoding chart. Exact codes of these columns and rows do not influence the number of compatible classes of the hyperfunction.

To construct a hyperfunction with better decomposability, we thus can take the same encoding strategy to encode ingredients as that used in compatible class encoding.

B. Decomposition of Hyperfunction

Methods for single-output decomposition can be easily applied to decompose a hyperfunction. Therefore, via hyperfunction decomposition, any algorithm proposed for single-output decomposition can be easily extended for multiple-output decomposition. Before detailing the process, we give the following definitions.

Definition 4: The **transitive fanout** of a node j, denoted as $\boldsymbol{TFOj}$, is defined as $TFOj = \{\text{node } i \mid i = j \text{ or } \exists \text{ path from } j \text{ to } i\}$.

Definition 5: The **duplication source *(DS)*** after a hyperfunction decomposition is defined as $DS = \{\text{node } j \mid j \text{ has at least one pseudo primary input as its direct fanin}\}$.

Note that after we have decomposed a hyperfunction, every node $\notin DS$ must be k-feasible. However, every node $\in DS$ with t pseudo primary inputs as its direct fanins must be $(t+k)$-feasible.

Definition 6: The **duplication cone *(DC)*** after a hyperfunction decomposition is defined as $DC = \{\cup_j TFOj \mid j \in DS\}$.

Definition 7: The $\boldsymbol{m}$th **layer duplication set** $(\boldsymbol{D}\mathbf{Set}_{\boldsymbol{m}})$ after a hyperfunction decomposition is defined as $D\text{Set}_m = \{\text{node } j \mid j \text{ is in } \textit{TFOs} \text{ of } m \text{ pseudo primary inputs}\}$.

After we have decomposed a hyperfunction, the DC should be duplicated to implement its ingredients. Assume the hyperfunction has n pseudo primary inputs and i ingredients. A node $\in D\text{Set}_m$ $(m < n)$ must be duplicated $(2^m - 1)$ additional copies; a node $\in D\text{Set}_n$ must be duplicated to have additional $(i-1)$ copies. To implement each ingredient, we then assign its corresponding code to the pseudo primary inputs. These pseudo primary inputs, assigned with constant values, can be collapsed into their fanout nodes. Nodes $\in DS$ are thus reduced by eliminating the extra pseudo primary inputs. After hyperfunction decomposition, all new generated nodes $\notin DC$ can be shared by these ingredients. Consider the following example.

Example 3: Assume that four distinct Boolean functions, $f_0(i_0, i_1, i_2, i_3, i_4, i_5, i_7, i_8)$, $f_1(i_0, i_1, i_2, i_3, i_4, i_5, i_6)$, $f_2(i_0, i_1, i_2, i_3, i_4, i_5)$, and $f_3(i_0, i_1, i_2, i_3, i_4, i_5)$, form a hyperfunction F: $B^{11} \rightarrow B$. Suppose the coding of each ingredient is derived by applying the compatible class encoding algorithm and is shown in Fig. 8(a). To implement F with five-input LUTs, assume that F is decomposed as shown in Fig. 8(b). Nodes in the duplication cone are filled with gray. All other nodes can be shared by the ingredients of F. After duplicating the duplication cone, as demonstrated in Fig. 9(a), we assign (η_0, η_1) as $(0, 0)$ to recover f_0, $(1, 0)$ to recover f_1, $(0, 1)$ to recover f_2 and $(1, 1)$ to recover f_3.

After we collapse these constant input signals in Fig. 9(a) into their fanout nodes, the resultant network will be independent of these pseudo primary inputs. Further in this example, since f_2 and f_3 are independent of i_6, i_7, and i_8 according to their original function expressions, we can further simplify the network by collapsing nodes a_1 and b_1 into f_2 and f_3 as shown in Fig. 9(b). Nodes $\notin$ duplication cone can be shared by the ingredients of F.

C. Properties of Hyperfunction Decomposition

Because nodes in the duplication cone must be duplicated, fewer nodes $\in DC$ are preferred. Consequently, we should keep pseudo primary inputs as close to the output as possible during the decomposition process. In other words, we prefer to keep pseudo primary inputs in the μ set during decomposition. In the extreme case, if we always select pseudo primary inputs in the μ set, hyperfunction decomposition can be considered as the *column encoding* [7]. Hence, the column encoding in [7] is a special case of our hyperfunction decomposition. Hyperfunction decomposition provides a more generic and flexible means to extract common sublogic. Since multiple-output functions can be decomposed as easily as a single-output function, the λ set size does not have to be restricted to a small value. Moreover, the shared logic can be across many levels; large common subexpressions can be extracted.

Although a hyperfunction is decomposed by applying single-output decomposition, two differences exist between hyperfunction decomposition and single-output decomposition. First, strict encoding for single-output decomposition may become nonstrict for hyperfunction decomposition because for each ingredient of a hyperfunction, a compatible class can be encoded with more than one code. (When a hyperfunction is constructed, conjunctive partitions may be performed on the partitions of ingredients. So a symbol in a partition may

$$F(\eta_0,\eta_1,i_0,i_1,i_2,i_3,i_4,i_5,i_6,i_7,i_8)$$
$$=\overline{\eta_0}\,\overline{\eta_1}f_0+\eta_0\overline{\eta_1}f_1+\overline{\eta_0}\eta_1 f_2+\eta_0\eta_1 f_3$$
$$f_0=F(0,0,i_0,i_1,i_2,i_3,i_4,i_5,i_6,i_7,i_8)$$
$$f_1=F(1,0,i_0,i_1,i_2,i_3,i_4,i_5,i_6,i_7,i_8)$$
$$f_2=F(0,1,i_0,i_1,i_2,i_3,i_4,i_5,i_6,i_7,i_8)$$
$$f_3=F(1,1,i_0,i_1,i_2,i_3,i_4,i_5,i_6,i_7,i_8)$$

(a)

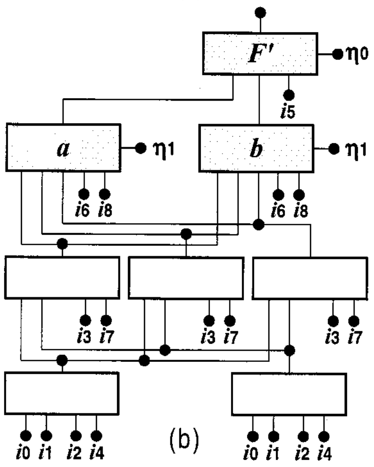


Fig. 8. (a) Hyperfunction and ingredients and (b) decomposition of F.

be broken into several symbols in a conjunctive partition.) Secondly, an encoding that is rigid for single-output decomposition may become pliable for hyperfunction decomposition because the compatible classes of an ingredient may be encoded by more bits than necessary. IMODEC [8] provides a good approach to guarantee that each function is rigidly encoded. However, pliable encoding could save more areas than rigid one in the cases discussed below.

Definition 8: A partition A is **contained** by another partition B if the multiplicity of B equals the multiplicity of the conjunctive partition of $\{A, B\}$.

Theorem 6: Given two partitions A of function f_a and B of function f_b with respect to the same λ set selection, both $\lceil\log_2(\text{multiplicity of } A)\rceil$ and $\lceil\log_2(\text{multiplicity of } B)\rceil$ are less than the λ set size. A is contained by B if and only if the decomposition functions of f_b (which identify the column patterns in B by strict encoding) can be used as the decomposition functions of f_a.

Theorem 7: Given two partitions A of function f_a and B of function f_b with respect to the same λ set selection both $\lceil\log_2(\text{multiplicity of } A)\rceil$ and $\lceil\log_2(\text{multiplicity of } B)\rceil$ are less than the λ set size. If A is contained by B, then the decomposition functions of f_b can be used as the decomposition functions of f_a.

Example 4: Given three functions $f_0(x_0,x_1,x_2,x_3,y_0,y_1)$, $f_1(x_0,x_1,x_2,x_3,y_2,y_3)$, and $f_2(x_0,x_1,x_2,x_3,y_3,y_4)$ with λ set selection as $\{x_0,x_1,x_2,x_3\}$, assume we have three partitions:

$$\Pi_0=\langle 0,0,1,0,1,2,2,0,3,2,0,0,0,0,0,2\rangle \text{ of } f_0$$
$$\Pi_1=\langle 0,1,2,0,2,3,3,2,4,3,0,2,1,5,1,3\rangle \text{ of } f_1$$
$$\Pi_2=\langle 0,1,1,0,1,2,2,3,3,2,0,3,1,4,5,2\rangle \text{ of } f_2.$$

If f_1 and f_2 are combined to construct a hyperfunction h_{12} with λ set $\{x_0,x_1,x_2,x_3\}$, then the hyperfunction has partition Πc of $\{\Pi_1,\Pi_2\}$, $\Pi c_{12}=\langle 0,1,2,0,2,3,3,4,5,3,0,4,1,6,7,3\rangle$. Because Πc of $\{\Pi_0,\Pi_1,\Pi_2\}$, Πc_{012}, has the same multiplicity as Πc_{12}, Π_0 is contained by Πc_{12} by Definition 8. According to Theorem 6 or Theorem 7, the decomposition functions of h_{12} can be used as the decomposition functions of f_0. Therefore, if f_0, f_1, and f_2 are combined to form a hyperfunction with λ set $\{x_0,x_1,x_2,x_3\}$, there are three decomposition functions (because of the multiplicity of $\Pi c_{012}=8$) shared by the three functions as shown in Fig. 10(a). Because f_0 uses three decomposition functions instead of two decomposition functions to encode four compatible classes, the encoding becomes pliable. On the other hand, if the encoding is restricted to being rigid, such as [8], it may derive the result as shown in Fig. 10(b). In this case, two more LUTs are consumed.

V. Experimental Results

Our algorithms have been implemented in the SIS environment [16]. Table I compares our new encoding algorithm with random encoding. The benchmark circuits (in the first column) are collapsed into two-level logic. Without any pre- or post-processing, we then directly apply single-output decomposition, with both random and new encoding approaches, on these benchmarks. The decomposed benchmark circuits are mapped into five-input and one-output LUTs. Experiments are run under a SUN SPARC 20 workstation. The results of random encoding are shown in column 2 with consumed CPU time in column 3; the results of our new encoding are shown in column 4 with CPU time in column 5. As can be seen, in the majority of instances, random encoding and new encoding generate almost the same results for small functions. In contrast, larger functions are better improved. Recall that the proposed encoding algorithm aims at simplifying the image function. As nodes under decomposition are not large enough, there is almost no room for improvement. Therefore, for multilevel circuits (nodes in Boolean networks may not be large enough), random encoding and our encoding produce almost the same results. Nevertheless, as our proposed encoding method suitable for large functions, it would be beneficial to exploit this method on hyperfunction decomposition because a hyperfunction is constructed by the combination of several multiple-output functions and could thereby be large.

The overall technology mapping method, HYDE, is conducted over a set of benchmark circuits. To prepare the initial circuits for the following technology mapping, small circuits are collapsed while large circuits are optimized by SIS algebraic script. (Benchmark circuit *des* is, in addition, partially collapsed such that several nodes can share the same supports.) After the initial circuits are derived, the technology mapping script used for two-level circuits is: our decomposition, xl_partition-tm, xl_cover and the script for multilevel circuits is: (full_)simplify, our decomposition, xl_partition-tm, xl_cover. For multilevel circuits, the script is applied several times to improve results by taking advantage of extracting the local don't care set. We compare experimental results with state-of-the-art FPGA synthesis techniques [7], [8], and [12] in Tables II and III.

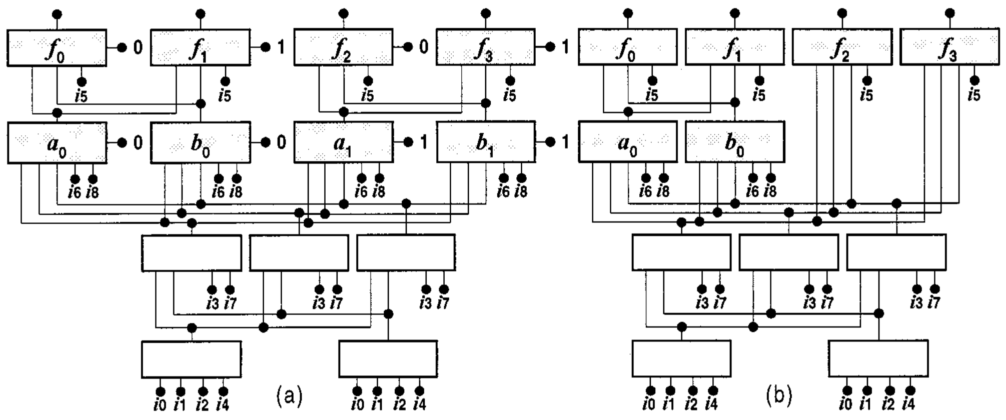


Fig. 9. (a) Duplication for ingredients and (b) further reductions.

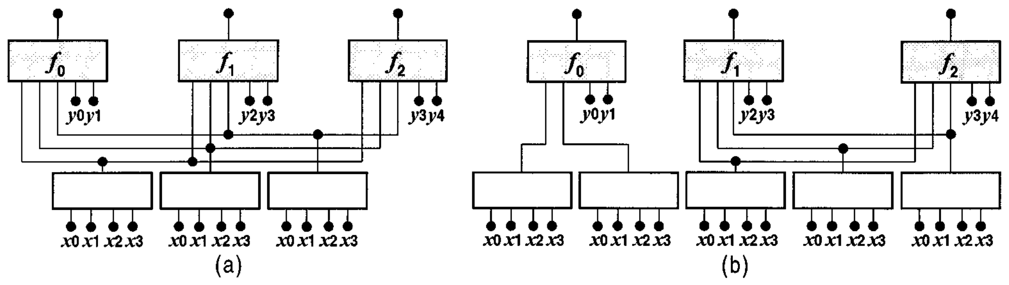


Fig. 10. (a) Pliable and (b) rigid encoding of f_0.

TABLE I
COMPARISON OF RANDOM ENCODING AND OUR NEW ENCODING

Circuit	Rand. Enc. LUT	CPU Time Sec.	New Enc. LUT	CPU Time Sec.
5xp1	20	1.3	20	1.3
9sym	7	11.7	6	11.9
alu2	69	56.2	69	57.0
apex4	392	215.8	392	222.4
b9	65	20.9	65	21.6
clip	27	19.5	27	19.1
count	76	32.0	76	33.0
duke2	995	528.4	941	465.8
e64	559	73.5	559	73.1
f51m	17	2.3	17	2.5
misex1	17	0.8	17	1.1
misex2	48	7.5	48	7.4
misex3	1661	1019.5	1526	3161.3
rd73	8	2.5	8	2.5
rd84	13	13.2	13	13.2
sao2	28	28.0	28	29.6
vg2	466	228.5	419	227.0
z4ml	6	1.1	6	1.4
Total	**4474**	**2262.7**	**4237**	**4351.2**

In Table II, the target architecture is the Xilinx XC3000 FPGA. Our algorithm outperforms IMODEC [8] and FGSyn [7] in most of the examples. Nevertheless, FGSyn produces much better result in circuit *alu4* than HYDE and IMODEC. In Table III, the resultant circuits are constructed by five-input one-output LUTs. Without much difference in the consumed CPU time, it is not shown in Table III. In columns 2–4, we repeat the results reported in [12]. Our algorithm does not map *alu4* well. However, the results are still better than those appearing in column 3. Exclusive of *alu4*, our algorithm produces slightly better results than those in column 4. Due to the disability of handling large circuits such as *C880* in [12], our approach is considered more practical to handle large circuits.

TABLE II
EXPERIMENTAL RESULTS FOR XC3000 DEVICE

Circuit	**IMODEC [8]**	**FGSyn [7]**	**HYDE**	**CPU Time**
	CLB	**CLB**	**CLB**	**Sec.**
5xp1	9	9	10	1.3
9sym	7	7	6	22.8
alu2	46	55	43	554.4
alu4	168	56	140	911.7
apex6	129	181	135	108.7
apex7	41	43	39	9.6
clip	12	18	11	407.2
count	26	23	24	1.6
des	489	-	408	236.6
duke2	122	85	75	28.0
e64	55	44	48	0.0
f51m	8	8	8	10.4
misex1	9	8	9	11.8
misex2	21	22	22	3.3
rd73	5	5	5	3.0
rd84	8	8	7	16.0
rot	127	136	125	132.7
sao2	17	25	17	117.5
vg2	19	17	18	3.6
z4ml	4	4	4	2.7
C499	50	54	50	2.9
C880	81	87	68	69.8
Total	**1453**		**1272**	
Subtotal	**964**	**895**	**864**	

TABLE III
EXPERIMENTAL RESULTS FOR FIVE-INPUT ONE-OUTPUT LUTS

Circuit	**w/o Resub. [12]**	**w/ Resub. [12]**	**Resub. PO [12]**	**HYDE**
	LUT	**LUT**	**LUT**	**LUT**
5xp1	15	11	10	13
9sym	7	7	7	6
alu2	48	48	48	50
alu4	172	90	56	206
apex4	374	374	374	354
apex6	192	161	155	186
apex7	120	61	54	54
b9	53	39	37	36
clip	18	11	14	14
count	52	31	31	31
des	-	-	-	561
duke2	175	155	150	116
e64	-	-	-	80
f51m	12	10	8	12
misex1	12	10	10	13
misex2	40	36	36	29
misex3	195	213	120	131
rd73	8	6	6	6
rd84	12	7	8	9
rot	-	-	-	185
sao2	23	21	21	22
vg2	44	21	17	18
z4ml	6	5	4	5
C499	-	-	-	70
C880	-	-	-	81
Total	**1578**	**1317**	**1166**	**1311**
Subtotal (– alu4)	**1406**	**1227**	**1110**	**1105**

VI. Conclusion

Compatible class encoding and hyperfunction decomposition techniques have been proposed. The former improves the decomposability of the image function and the hyperfunction, while the latter extracts common subexpressions among multiple-output functions. By transforming multiple-output functions into a single-output hyperfunction, the problem of multiple-output decomposition can thus be reduced to that of single-output decomposition. As a result, previous efforts intended for single-output decomposition can be easily extended to solve multiple-output decomposition. Experimental results show that our encoding method performs well for large nodes. In contrast, it does not provide much improvement for small ones. Fortunately, our encoding method is suitable for hyperfunctions, which are combined from small functions and eventually could be large. The overall technology mapping is promising.

For future research, it will be more powerful if compatible class encoding simultaneously considers the image and decomposition functions. On the other hand, to use the hyperfunction decomposition technique for common sublogic extraction in technology independent logic synthesis, it would be relatively time consuming and impractical. Further research is necessary to make it efficient for this extension.

References

[1] R. L. Ashenhurt, "The decomposition of switching functions," *Ann. Computation Lab. Harvard Univ.*, vol. 29, pp. 74–116, 1959.

[2] J. P. Roth and R. M. Karp, "Minimization over Boolean graphs," *IBM J.*, pp. 227–238, 1962.

[3] R. M. Karp, "Functional decomposition and switching circuit design," *J. Soc. Indust. Appl. Math.*, vol. 11, no. 2, pp. 291–335, 1963.

[4] W.-Z. Shen, J.-D. Huang, and S.-M. Chao, "Lambda set selection in Roth–Karp decomposition for LUT-based FPGA technology mapping," in *Proc. ACM/IEEE Design Automation Conf.*, June 1995, pp. 65–69.

[5] J.-H. Jiang, J.-Y. Jou, J.-D. Huang, and J.-S. Wei, "BDD based lambda set selection in Roth–Karp decomposition for LUT architecture," in *Proc. Asia South Pacific Design Automation Conf.*, Jan. 1997, pp. 259–264.

[6] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimum functional decomposition using encoding," in *Proc. ACM/IEEE Design Automation Conf.*, June 1994, pp. 408–414.

[7] Y.-T. Lai, K.-R. R. Pan, and M. Pedram, "OBDD-based function decomposition: Algorithms and implementation," *IEEE Trans. Comput.-Aided Design*, vol. 15, pp. 977–990, Aug. 1996.

[8] B. Wurth, K. Eckl, and K. Antreich, "Functional multiple-output decomposition: Theory and an implicit algorithm," in *Proc. ACM/IEEE Design Automation Conf.*, June 1995, pp. 54–59.

[9] J.-D. Huang, J.-Y. Jou, and W.-Z. Shen, "Compatible class encoding in Roth–Karp decomposition for two-output LUT architecture," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 359–363.

[10] C. Legl, B. Wurth, and K. Eckl, "Computing support-minimal subfunctions during functional decomposition," *IEEE Trans. VLSI Syst.*, vol. 6, pp. 354–363, Sept. 1998.

[11] J. Cong and Y.-Y. Hwang, "Partially-dependent functional decomposition with applications in FPGA synthesis and mapping," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, Feb. 1997, pp. 35–42.

[12] H. Sawada, T. Suyama, and A. Nagoya, "Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 353–358.

[13] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computers*, pp. 677–691, Aug. 1986.

[14] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-Level Synthesis*: Kluwer, 1992.

[15] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. New York: Wiley, 1988.

[16] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. Comput.-Aided Design*, pp. 1062–1081, Nov. 1987.

Jie-Hong Jiang received the B.S. and M.S. degrees in electronics engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1996 and 1998, respectively. He is currently working toward the Ph.D. degree at the University of California, Berkeley.

During the compulsory military service, from 1998 to 2000, he joined the Taiwan Air Force as a 2nd Lieutenant.

Mr. Jiang is a member of Phi Tau Phi.

Jing-Yang Jou (M'96) received the B.S. degree in electrical engineering from the National Taiwan University and the M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign.

He is a Professor and Chairman in the Department of Electronics Engineering at National Chiao Tung University, Taiwan. He has worked in the GTE Laboratories and Bell Laboratories. He has published more than 80 journal and conference papers. His research interests include behavioral and logic synthesis, VLSI designs and CAD for low power, design verification, and hardware/software codesign.

Dr. Jou is a member of Tau Beta Pi and the recipient of the Distinguished Paper Award of the IEEE International Conference on Computer-Aided Design, 1990. He served as the Technical Program Chair of the Asia-Pacific Conference on Hardware Description Languages (APCHDL'97).

Juinn-Dar Huang received the B.S. and Ph.D. degrees in electronics engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1992 and 1998, respectively.

He is currently a Research and Development Deputy Manager with the Global UniChip Corporation, Hsinchu, Taiwan. His current research interests include logic synthesis, design verification, reusable IP authoring, and SoC design methodology.

Dr. Huang is a member of Phi Tau Phi.