VLSI Design
1994, Vol. 2, No. 2, pp. 89–103
Reprints available directly from the publisher
Photocopying permitted by license only

Technology Mapping for FPGA Using Generalized Functional Decomposition*

KUO-HUA WANG and CHENG CHEN
Department of Computer Science and Information Engineering, National Chiao Tung University, HsinChu, Taiwan 30050, R.O.C.

TING TING HWANG
Department of Computer Science, National Tsing Hua University, HsinChu, Taiwan 30043, R.O.C.

In this paper, we address the technology mapping for RAM-based FPGA. Functional decomposition is applied to decompose a large function into a set of smaller subfunctions such that each subfunction can be implemented using a single logic cell. Our system is mainly divided into two parts. The first part is designed specifically for totally symmetric functions. A *Fast-Decompose* algorithm based on weight dependency is proposed. The second part deals with general functions. We consider some techniques such as *output partition, variable partition, don't care assignment* and *encoding* to minimize the number of subfunctions derived. Using these techniques together, our tool, *Fun-Map,* improves the mapping results compared with other tools in terms of area and delay.

Key Words: *Field Programmable Gate Arrays; Functional decomposition; Complex disjoint decomposition; Variable partition; Output partition; Totally symmetric functions.*

1 INTRODUCTION

Field Programmable Gate Arrays (FPGA's) consist of arrays of programmable logic blocks and programmable routing networks. It provides both fast turn-around time and user-programmability for ASIC design. There are mainly two types of FPGA architecture: one is lookup table (RAM) based (e.g., AT&T, Xilinx), and the other is multiplexer based (e.g., Actel). For RAM based architecture, a novel feature of these devices is that a basic logic cell can implement any Boolean functions that satisfy the I/O constraints of the logic cell. In this paper, we concentrate on the technology mapping for the RAM based FPGA's.

Technology mapping is a process of transforming a technology independent Boolean network into a technology dependent network. Traditional techniques for technology mapping use a restricted set of cell library [6, 7]. These techniques are not suitable for mapping RAM-based FPGA architecture since a single logic cell can emulate too many functions. For example, for Xilinx 3000 series, a logic cell can implement 2^{2^5} different functions. Recently, many technology mapping systems for RAM-based FPGA have been developed [12–25].

The functional decomposition theory developed by Ashenhurst [1] have been used for designing switching circuits [4]. It was also used for PLA decomposition [5, 8, 30, 32, 33] and multilevel logic synthesis [9, 26]. In this paper, we apply functional decomposition to RAM-based FPGA technology mapping. Functional decomposition decomposes a function considering the functionality rather than the given network.

A totally symmetric function is one in which each of the input variables plays the same role in determining the value of the function. Based on the weight dependency and functional decomposition, a fast algorithm using full-adders to map totally symmetric functions is also proposed.

The rest of this paper is organized as follows. In the next section, the functional decomposition for FPGA design is presented. Section 3 introduces overview of the technology mapping system. In Section 4, a weight based algorithm specifically for handling

*This work was supported by the National Science Council, R.O.C., under contract no. NSC 81-0404-E-007-610.

totally symmetric functions is proposed. In Section 5, several techniques for general function decomposition such as output partition, variable partition, don't care assignment and encoding are considered. In Section 6, experimental results are presented. Lastly, a brief conclusion is given.

2 FUNCTIONAL DECOMPOSITION AND FPGA MAPPING

2.1 Simple and Complex Decompositions

Definition 2.1 Given a Boolean function $f(X)$. X and f are the input set and the output set, respectively. Then $\pi = (X_1, X_2)$ is a *partition* of X if $X_1 \cap X_2 = \emptyset$ and $X_1 \cup X_2 = X$. □

Definition 2.2 Let $f(X)$ be a multi-output function. Then $|f|$ is defined as the number of outputs. $|X|$ is defined as the number of inputs. □

Definition 2.3 Let $X = \{x_1, x_2, \ldots, x_k\}$ be an input set. Define $BS(X) = \{(\beta_1, \beta_2, \ldots, \beta_k) | \beta_i = 0 \text{ or } 1, i = 1 \cdots k\}$ as the Boolean space spanned by X. □

Definition 2.4 Given a function $f(X)$ and a partition $\pi = (X_1, X_2)$. Let $b_1, b_2 \in BS(X_1)$. b_1 and b_2 are *compatible* with respect to π, if $f(b_1, c) = f(b_2, c)$ for $\forall c \in BS(X_2)$; otherwise, they are *incompatible*. It is denoted as $b_1 \sim (\not\sim) b_2 / f$. □

This compatibility divides $BS(X_1)$ into many equivalent classes. For any two elements in the same equivalent class, they are compatible.

Definition 2.5 Given a function $f(X)$ and a partition $\pi = (X_1, X_2)$. $EQU(f, \pi)$ is defined as the number of equivalent classes in $BS(X_1)$ with respect to π. □

Example 2.1 Let $f(a, b, c, d) = (\bar{a}\bar{b} + ab)c + (\bar{a}b + a\bar{b})\bar{d}$ and a partition $\pi = (\{a, b\}, \{c, d\})$. Since $f_{\bar{a}\bar{b}} = f_{ab} = c$ and $f_{\bar{a}b} = f_{a\bar{b}} = \bar{d}$, $00 \sim 11$ and $01 \sim 10$ with respect to π. And $EQU(f, \pi) = 2$. □

Consider a Boolean function $f(X)$ and a partition $\pi = (X_1, X_2)$. The decomposition of f with respect to π under consideration is

$$f(X) = f_t(f_e(X_1), X_2), \tag{1}$$

where f, f_t and f_e are all multiple output functions. The lower bound on the number of encoding functions, $|f_e|$, depends on the number of equivalent classes in $BS(X_1)$. That is, $|f_e| \geq \log_2$ (the number of equivalent classes in $BS(X_1)$) [28].

If the number of equivalent classes in $BS(X_1)$ is less than or equal to 2, it is a simple disjoint decomposition; otherwise, it is a complex disjoint decomposition [4]. These two types of decompositions are shown in Figure 1.

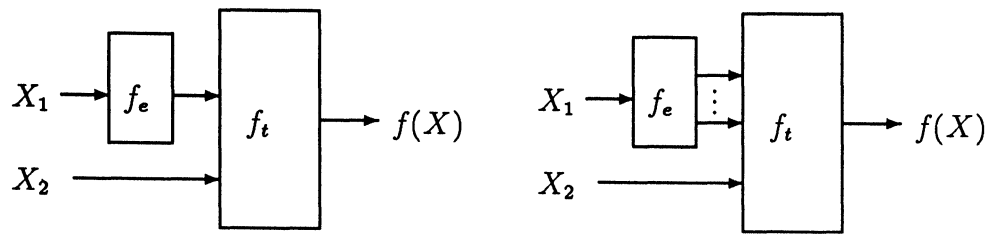
Using functional decomposition to decompose $f(X)$, two problems should be considered: (1) partition problem: to find a good partition π; (2) encoding problem: to encode the equivalent classes.

For the first problem, a partition with minimal number of equivalent classes should be found to minimize the number of interconnections between f_e and f_t. For the encoding problem, f_e can be found by assigning different binary codes to the equivalent classes in $BS(X_1)$. However, an assignment which minimizes overall cost of realizing the functions f_e and f_t needs to be selected.

2.2 FPGA Mapping with Complex Disjoint Decomposition

Simple disjoint decomposition handles only a very limited set of functions. For more general functions, we consider the complex disjoint decomposition.

Definition 2.6 The *support* of $f(X)$ is defined as the set of variables which f explicitly depends on. It is



(a)Simple Disjoint Decomposition. (b)Complex Disjoint Decomposition.

FIGURE 1 The functional decomposition of $f(X)$ by $\pi = (X_1, X_2)$.

denoted as $sup(f)$. The $sup(f)$ can be obtained by eliminating variables v which results in $f_v = f_{\bar{v}}$. □

Definition 2.7 Given a function f and an integer $k > 0$. Then f is *feasible* with respect to k if $|sup(f)| \leq k$; otherwise, it is *infeasible.* □

Recall that a logic cell can implement any k-input functions. If a function $f(X)$ has $|sup(f)| > k$, it can not be realized by a single logic cell directly. To map an infeasible function into logic cells, functional decomposition can be used to decompose it into many feasible subfunctions.

From Equation (1), we know that the derived subfunctions are f_t and f_e. In order to make f_e feasible, the partition $\pi = (X_1, X_2)$ used for decomposition must have $|X_1| \leq k$. Then each output of f_e can be implemented by a single logic cell. Similarly, to implement f_t, the size of input set, $X_t = X_2 \cup f_e$, of f_t must be less than k; otherwise, f_t must be further decomposed until all subfunctions are feasible.

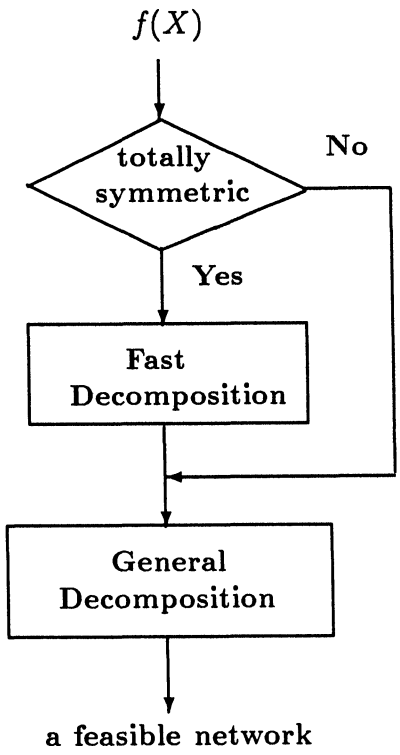


FIGURE 2 The flowchart of *Fun-Map.*

3 SYSTEM OVERVIEW

In this section, our functional decomposition based technology mapping system is overviewed. The system is mainly divided into two parts. The first part is designed specifically for totally symmetric functions; the second part uses the decomposition technique to decompose general functions. The flowchart of this technology mapping system is shown in Figure 2.

The system proceeds by first removing the functions which can be implemented using a single CLB. Then, the remaining functions are checked to see if there is any totally symmetric function. The symmetry is checked by examining on-set and off-set of the function, f [10]. Characteristic set returns the symmetry character of f. It gives the number of 1's among variables of X in on-set of f. A fast decomposition is then performed on totally symmetric functions using full-adders to synthesize subfunctions. This *Fast-Decompose* procedure will be discussed in detail in Section 4.

The general functions are handled in the second part. Some techniques such as output partition, variable partition, don't care assignment and encoding are considered. Output partition is to partition primary outputs into several groups based on some criterion. Then each group is mapped individually. Variable partition is to find the "best" partition of input variables for a completely specified function. For incompletely specified functions, a variable partition and don't care assignment are considered together. To derive subfunctions f_t and f_e, we need encode the subfunctions f_e. In many cases, the encoding may generate don't care set to f_t. This don't care set is very useful in the next level decomposition. These techniques are presented in Section 5. The functional decompsoition algorithm used in the system is described in Figure 3.

4 A MAPPING ALGORITHM FOR TOTALLY SYMMETRIC FUNCTIONS

4.1 Symmetric Functions

A function $f(X)$ is said to be symmetric with respect to a set $\lambda \subseteq X$, if and only if it is invariant under any permutation of variables in λ. λ is called the *symmetry set* of f. If $\lambda = X$, it is *totally symmetric*; otherwise, it is *partially symmetric*. For the case $|\lambda| = 2$, f is *pairwise symmetric* with respect to the *symmetry pair* λ.

For a totally symmetric function f, there is a simpler form to represent it. It can be specified using the number of 1's needed for the function to be 1. This is stated formally in the following theorem [29].

```
    Procedure Fun-Map(f(X)))
1   f = Remove(f); /* remove feasible outputs */
2   if f is totally symmetric then {        /* PART 1 */
        A_i's = Characteristic-Set(f(X));
        f = Fast-Decompose(f, X, A_i's);
        f = Remove(f);
    }
3   if | f |> 0 then {        /* PART 2 */
        f_1, f_2, ..., f_m = Output-Partition(f, X);
        for i = 1 to m do {
            if f_i is completely specified then {
                π = (X_1, X_2) = Variable-Partition(f_i, X);   /* | X_1 | ≤ k */
            } else {
                π = (X_1, X_2) = DC-Variable-Partition(f_i, X);
            }
            f_t, f_e = Encoding(f_i, X, π);   /* f_i(X) = f_t(f_e(X_1), X_2) */
            X_t = f_e ∪ X_2;
            Fun-Map(f_t(X_t));
        }
    }
4   end.
```

FIGURE 3 The *Fun-Map* procedure.

Theorem 4.1 $f(X)$ is totally symmetric if and only if it can be specified by a set $A = \{a_1, a_2, \ldots, a_m\}$, $0 \leq a_i \leq n$ where $n = |X|$ such that $f = 1$ if and only if exactly a_i of the input variables are 1. The function f is then denoted as S_A^n. □

Definition 4.1 For a totally symmetric function $f = S_A^n$, set A is the *characteristic set* of f. □

Example 4.1 For the three-variable majority function, $f = x_1x_2 + x_2x_3 + x_1x_3$, $A = \{2, 3\}$; for the two-variable parity function, $f = x_1\bar{x}_2 + \bar{x}_1x_2$, $A = \{1\}$. □

Since the value of $f(x_1, x_2, \ldots, x_n) = S_A^n$ depends on the characteristic set A, it can be also specified by an equation as follows:

$$\begin{cases} x_1 + x_2 + \cdots + x_n = a_i \\ \text{for } a_i \in A \subseteq \{0, 1, \ldots, n\}. \end{cases} \tag{2}$$

It can be viewed as a kind of threshold logic in which each input x_i is assigned a weight 1. For any input vector α, if Equation (2) is evaluated to be true then $f(\alpha) = 1$; otherwise $f(\alpha) = 0$.

Example 4.2 Consider the three-variable majority function $f = S_{\{2,3\}}^3$. It is specified by the following equation:

$$\begin{cases} x_1 + x_2 + x_3 = a_i \\ \text{for } a_i \in \{2, 3\}. \end{cases}$$

Based on a theorem proposed in [11], symmetry sets of a function f can be detected by first finding all symmetry pairs of f and then using these pairs to form larger symmetry sets. It is clear that if variables x_i and x_j form a symmetry pair of f then $f = f_{x_i \leftrightarrow x_j}$, where $f_{x_i \leftrightarrow x_j}$ is obtained from f by exchanging the variables x_i and x_j. We have proposed an efficient transpositional operation [35] for computing $f_{x_i \leftrightarrow x_j}$ from f in OBDD (Ordered Binary Decision Diagram) representation [2]. With f and $f_{x_i \leftrightarrow x_j}$ represented in OBDD, the equivalence checking, $f = f_{x_i \leftrightarrow x_j}$, can be performed in constant time. Using this method, it is relatively easy to detect large symmetry sets of completely specified functions. We also generalize this method for incompletely specified functions.

4.2 A Weight Based Algorithm

The following lemma suggests that a totally symmetric function is especially suitable for functional decomposition.

Lemma 4.2 Given a function $f(X)$. Let set $X_s \subset X$ be a symmetry set of f. Then the number of equivalent classes in $BS(X_s)$ is not greater than $k + 1$, where $|X_s| = k$.

Proof: The Boolean space of X_s, $BS(X_s)$, is divided into many classes depending on the number of 1's in variables of X_s. That is, any elements (minterms) with the same number of 1's in the variables of X_s are in the same equivalent class. Counting the number of 1's in variables of X_s, there are at most $k + 1$ cases. Therefore, the number of equivalent classes in $BS(X_s)$ is not greater than $k + 1$. □

For a totally symmetric function $f(X)$ and a subset $X_s \subset X$, where $|X_s| = k$, we know that there is at most $k + 1$ equivalent classes in $BS(X_s)$. Compared to general functions where the number of equivalent classes may be 2^k, totally symmetric functions are especially suitable for functional decomposition. For $k + 1$ equivalent classes, we need at least $\lceil \log_2 (k + 1) \rceil$ subfunctions to encode the information concerning X_s. Since there exists exponential number of different encodings in terms of encoding length [28] and the synthesized circuit depends on the encoding very much, we need to derive an effective and efficient encoding algorithm.

In order to retain the symmetric property of the function, we propose the following encoding method. Consider a totally symmetric function $f(X)$ with characteristic set A and a subset $X_s \subset X$, where $|X_s| = k$. First, we partition the elements in $BS(X_s)$ into $k + 1$ groups according to the number of 1's in an element. The elements in the same group have the same number of 1's and are thus in the same equivalent classes. The weight of a group is the number of 1's in an element of the group. Then, the group with weight i is encoded as the binary value i. Figure 4 shows the encoding for the case of $|X_s| = 4$. Three subfunctions e_0, e_1 and e_2 are derived. For such an encoding, each bit corresponds to a weight of power of 2. For the example shown in Figure 4, e_0, e_1, e_2 correspond to the weight of 2^0, 2^1, 2^2, respectively.

Now we partition inputs into m sets $X_{s_1}, X_{s_2}, \ldots, X_{s_m}$, where $X_{s_1} \cup X_{s_2} \cdots X_{s_m} = X$ and $X_{s_i} \cap X_{s_j} = \emptyset$ for $i \neq j$. Encode each set of inputs as described above. We will have a structure as shown in Figure 5.

$x_1x_2x_3x_4$		$e_2e_1e_0$
0000		000
0001	0010	001
0100	1000	
0011	0101	010
0110	1001	
1010	1100	
0111	1011	011
1101	1110	
1111		100

FIGURE 4 The encoding for $n = 4$.

Let's denote the encoded outut bits for input set X_{s_m} as e_{m_i} for $0 \leq i \leq l$, where $l + 1$ is the encoding length and i represents 2^i weight for the input set X_{s_m}. By grouping the encoded outputs with the same weight together, the totally symmetric function $f(X) = f_t(e_{i_j}'s)$ and f_t can be specified as

$$\begin{cases} 2^l(e_{1_l} + e_{2_l} + \cdots + e_{m_l}) \\ \quad + \cdots + 2^0(e_{1_0} + e_{2_0} + \cdots + e_{m_0}) = a_i \\ \text{for } a_i \in A \subseteq \{0, 1, \ldots, n\}. \end{cases} \quad (3)$$

where $l + 1$ is the encoding length. Notice that f_t is a partially symmetric function with $l + 1$ symmetry sets $S_0, S_1, \ldots, S_l$. Each symmetry set $S_i = \{e_{1_i}, e_{2_i}, \ldots, e_{m_i}\}$ has a weight 2^i. Now for each set S_i, we define a function f_{t_i} as a function of S_i. f_{t_i} counts the number of 1's of $e_{1_i}, e_{2_i}, \ldots, e_{m_i}$. Thus the function f_{t_i} is also a totally symmetric function. The defined function f_{t_i} can be decomposed and encoded using the same method. Note that f_t is defined on f_{t_i} and the weight of f_{t_i} must be multiplied by 2^i. This

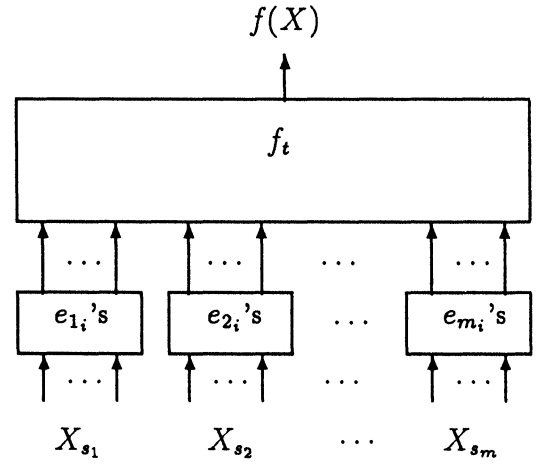


FIGURE 5 The top level decomposition of $f(X)$.

$$\overbrace{x_1 + x_2 + x_3}^{3} + \overbrace{x_4 + x_5 + x_6}^{3} + \overbrace{x_7 + x_8 + x_9}^{3} = a_i$$
$$\Downarrow$$
$$(2e_{1_1} + e_{1_0}) + (2e_{2_1} + e_{2_0}) + (2e_{3_1} + e_{3_0}) = a_i$$
$$\Downarrow$$
$$2(\overbrace{e_{1_1} + e_{2_1} + e_{3_1}}^{3}) + (\overbrace{e_{1_0} + e_{2_0} + e_{3_0}}^{3}) = a_i$$
$$\Downarrow$$
$$2(2E_{1_1} + E_{1_0}) + (2E_{2_1} + E_{2_0}) = a_i$$
$$\Downarrow$$
$$4E_{1_1} + 2(E_{1_0} + E_{2_1}) + E_{2_0} = a_i$$

FIGURE 6 The decomposition process of 9-input $f(X)$.

process can be applied recursively until the size of each symmetry set is less than a predefined number. The last stage function is then synthesized straightforward according to the weights that each output bits represents.

For example, consider a 9-input totally symmetric function $f(X)$ with a characteristic set, $A = \{3, 4, 5, 6\}$. The function is specified as

$$\begin{cases} x_1 + x_2 + \cdots + x_9 = a_i \\ \text{for } a_i \in \{3, 4, 5, 6\}. \end{cases}$$

Now, let the input limit of derived subfunctions be 3. First, partition X into 3 sets, $X_1 = \{x_1, x_2, x_3\}$, $X_2 = \{x_4, x_5, x_6\}$ and $X_3 = \{x_7, x_8, x_9\}$ such that the input size of each set X_i is smaller than or equal to the input limit. By our algorithm, each set X_i is encoded by 2 output bits e_{i_1}, e_{i_0}. Then $f(X) = f_t(e_{i_j}$'s) and f_t can be specified as

$$\begin{cases} 2(e_{1_1} + e_{2_1} + e_{3_1}) + (e_{1_0} + e_{2_0} + e_{3_0}) = a_i \\ \text{for } a_i \in \{3, 4, 5, 6\}. \end{cases}$$

Now f_t is partially symmetric and can be further decomposed until the size of each symmetry set is less than 3 (the predefined number). The complete decomposition process and the synthesized circuit are shown in Figure 6 and Figure 7, respectively. Notice that the last stage function f_t is specified as

$$\begin{cases} 4E_{1_1} + 2(E_{1_0} + E_{2_1}) + E_{2_0} = a_i \\ \text{for } a_i \in \{3, 4, 5, 6\}. \end{cases}$$

4.3 FPGA Technology Mapping

In the Xilinx 3000 series family, a Configurable Logic Block (CLB) can implement any single function of up to 5 variables, or any two-output function of up to 5 input variables and each output depending on at most 4 inputs. In order to take the advantage of the Xilinx architecture, we set the input size of subfunctions to be 3. Thus there are 4 groups in the Boolean space of an input set and the encoding length is two which can be implemented in a single CLB. By such a grouping, the two encoded bits become the sum bit and carry bit of a full-adder, and a CLB corresponds to a full-adder. The symmetric function is thus constructed using the full-adder as a basic block [36]. The decomposition algorithm is shown in Figure 8.

As shown in Figure 8, *Fast-Decompose* stops when the size of each symmetry set is less than 3 or function f_t is feasible. The time complexity of this procedure is equivalent to the number of full-adders used. In

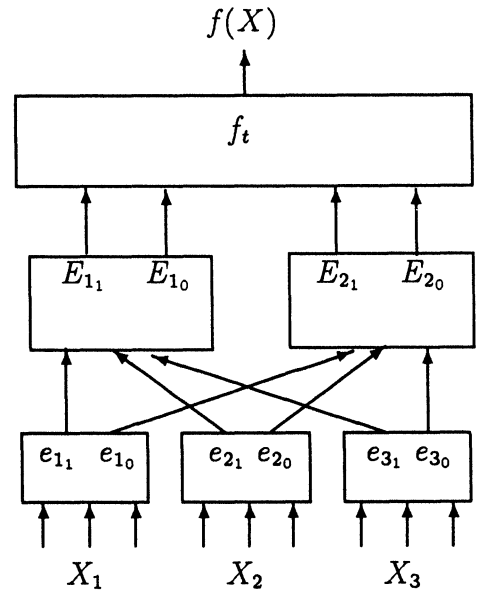


FIGURE 7 The synthesized circuit of 9-input $f(X)$.

```
Procedure Fast-Decompose(f, S0, Ai's)
/* Symmetry set S0 = {x1, x2, ..., xn}; */
/* Characteristic sets Ai's; */
/* Initially, Si = ∅ for i > 0 and m = 0. */

1 while there exists set Si and | Si |≥ 3 do {
2   for each set Si, where | Si |≥ 3, do {
      Partition Si into a set of disjoint groups Gj's and each | Gj |= 3;
3     for each group Gj do {
        Assign a full-adder Fm to Gj and build two internal variables sm and cm;
        /* sum and carry */
        Si = Si − Gj + {sm};
        Si+1 = Si+1 + {cm};
        m = m + 1;
4       if ft is feasible with respect to k then { /* k = 5 */
          End this decomposition; /* go to step 5 */
        }
      }
    }
  }
5 for each characteristic Set Ai do {
    Generate the function fti with the input variables in Si's;
  }
6 return(fti's).
```

FIGURE 8 The *Fast-Decompose* procedure.

[34], we have formally proved that the number of full-adders used for constructing a totally symmetric function $f = S^n_A$ is less than n, where n is the number of input variables. Therefore, the time complexity of *Fast-Decompose* is bounded by O(n).

5 A GENERAL DECOMPOSITION ALGORITHM

To handle the general function and the last stage function resulted from decomposition of symmetric function, we consider the complex disjoint decomposition for mapping general Boolean functions. In this section, we map the general Boolean functions to FPGA's. Some techniques are proposed to improve the mapping results. They are briefly described in the following.

(1) Output partition: partition the outputs into a set of groups and then each group is decomposed individually;

(2) Variable partition: find the "best" partition π with minimum number of equivalent classes for completely specified functions;

(3) Don't care assignment: assign the don't cares for minimizing the number of equivalent classes for incompletely specified functions;

(4) Encoding: derive subfunctions f_t, f_e and generate don't cares; non-decomposable functions are also handled separately.

5.1 Output Partition

For a given function $f(X)$ and a partition $\pi = (X_1, X_2)$, we say that f is *non-decomposable* with respect to π if $\lceil \log_2 m \rceil = |X_1|$, where m is the number of equivalent classes in $BS(X_1)$. In some cases, a function is non-decomposable if the multiple output function is treated as a whole while it is decomposable if each individual function is handled separately.

Example 5.1 Consider a 2-output function $f(a, b, c, d)$ consisting of f_1 and f_2. The decomposition

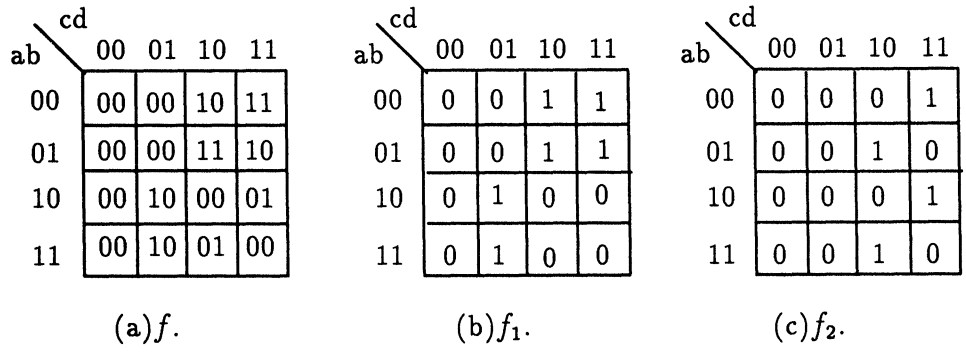


FIGURE 9 The decomposition charts of f, f_1 and f_2.

charts [4] of f, f_1, f_2 with respect to $\pi = (\{a, b\}, \{c, d\})$ are shown in Figure 9(a), (b) and (c), respectively. By observing their decomposition charts, we know that $EOU(f_1, \pi) = 2$, $EOU(f_2, \pi) = 2$ and $EOU(f, \pi) = 2 \cdot 2 = 4$. So both f_1 and f_2 are decomposable but f isn't. □

However, if we consider individual output separately, we will not be able to take subcircuits sharing into consideration.

From the above observations, we suggest that the outputs of f should be partitioned into several decomposable groups. Then each group is decomposed individually. We propose to partition outputs by grouping the outputs with 'similar' best partition. Usually, for a given partition, the best partition is not unique. We first find all good partitions for each individual output. Then partition outputs into several groups by checking if two outputs have the same good partition or almost the same partition. It is more likely that grouped functions are decomposable if they have the same good partition. Moreover, since the grouped outputs use the same input partition, it is more likely to have subcircuits sharing.

As described above, the time complexity of output partition is dominated by finding good input partitions for each output. Since finding the best input partition needs to search all possible input partitions, it is exponential in terms of input size. Instead of using exhaustive searching method, we propose a branch and bound algorithm for finding good input partitions which will be described in the following section. Therefore, the time complexity of output partition is O(np), where n is the number of outputs and p is the time complexity of the branch and bound algorithm.

5.2 Variable Partition

To find the best input partition, we need to define cost function for input partitions, and provide a procedure for computing the cost for a given partition. Based on the cost function, we also need an efficient algorithm for searching a best partition. In this section, we propose an easily computed cost function to estimate the number of equivalent classes of a given partition. A branch and bound algorithm is then developed to find the partition with least estimated cost among all partitions.

In [27], the number of equivalent classes for a given partition is computed using the number of different overlappings of cubes. An overlapping of cubes means the intersection of the cubes is not empty. Intuitively, less number of cube patterns will produce less number of cube overlappings and thus less number of equivalent classes. By such an observation, we consider a partition that has the smallest number of cube patterns as a best partition. Now, we define the cube patterns.

Definition 5.1 Let $C_f = \{c_1, c_2, \ldots, c_m\}$ be the onset of $f(X)$ and $\pi = (X_1, X_2)$ be a partition of X. Then cube pattern $Cube(C_f, X_1)$ is defined as $Cube(C_f, X_1) = \{l_i | \exists r_i, (l_i, r_i) \in C_f, i = 1, \ldots, m\}$ and $Cube(C_f, X_2)$ is defined as $Cube(C_f, X_2) = \{r_i | \exists l_i, (l_i, r_i) \in C_f, i = 1, \ldots, m\}$, where l_i and r_i are the coordinates associated with the variables in X_1 and variables in X_2, respectively. □

For a given function $f(X)$ and partition $\pi = (X_1, X_2)$, the cost function is defined as:

$$cost(f(X), \pi) = \min(|Cube(C_f, X_1)|, |Cube(C_f, X_2)|). \quad (4)$$

Example 5.2 Consider a comparison function

$$f(a_1, a_2, b_1, b_2) = \bar{a}_1\bar{a}_2\bar{b}_1\bar{b}_2 + \bar{a}_1a_2\bar{b}_1b_2 + a_1\bar{a}_2b_1\bar{b}_2 + a_1a_2b_1b_2$$

which compares 2-bit binary numbers $A = (a_1, a_2)$ and $B = (b_1, b_2)$. Assume we have two partitions $\pi_1 = (\{a_1, a_2\}, \{b_1, b_2\})$ and $\pi_2 = (\{a_1, b_1\}, \{a_2, b_2\})$. Then $Cube(f, \{a_1, a_2\}) = \{\bar{a}_1\bar{a}_2, \bar{a}_1a_2, a_1\bar{a}_2, a_1a_2\}$, $Cube(f, \{b_1, b_2\}) = \{\bar{b}_1\bar{b}_2, \bar{b}_1b_2, b_1\bar{b}_2, b_1b_2\}$, $Cube(f, \{a_1, b_1\})$

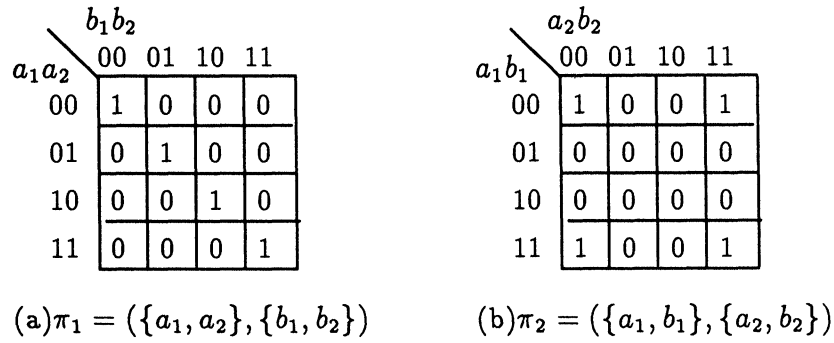


FIGURE 10 The decomposition charts of 2-bit comparison function.

$= \{\bar{a}_1\bar{b}_1, a_1b_1\}$ and $Cube(f, \{a_2, b_2\}) = \{\bar{a}_2\bar{b}_2, a_2b_2\}$. Thus $cost(f, \pi_1) = 4$ and $cost(f, \pi_2) = 2$. The decomposition charts of these two partitions are shown in Figure 10. It shows that the numbers of equivalent classes for π_1 and π_2 are 4 and 2, respectively. The partition π_2 has smaller cost and less number of equivalent classes than π_1. □

With this cost function, we now show a branch and bound algorithm for finding the partition with the least estimated cost.

Observation 5.1 Given a function $f(X)$ and two partitions $\pi_1 = (Y, X - Y)$ and $\pi_2 = (Z, X - Z)$, where $Y \subset Z \subseteq X$. Then $|Cube(C_f, Y)| \leq |Cube(C_f, Z)|$. □

From the cost function defined above, we know that the problem of finding a best partition $\pi = (X_1, X_2)$ can be solved by finding a subset $S \subset X$ and $|S| = \min(|X_1|, |X_2|)$ with minimum number of cube patterns. Without loss of generality, we let X_1 be the set of smaller size. Based on Observation 5.1, a branch and bound algorithm is developed for finding a subset $X_1 \subset X$ with the minimum number of cube patterns.

First, we find a best solution by including variables into X_1 one by one. Variables are included into X_1 in a greedy way. The best solution found is then used to bound the search space. We use an example to explain the branch and bound algorithm in more detail.

Consider the function $f = \bar{a}b(\bar{d}ef + df + \bar{e}\bar{f}) + \bar{b}c(\bar{e} + \bar{d}e + de\bar{f})$. To find a best partition $\pi = \{X_1, X_2\}$ with $|X_1| = 3$, the search tree is shown in Figure 11. In the search tree, the square nodes at level 3 are *solution* nodes and the circle nodes are the *ex-*

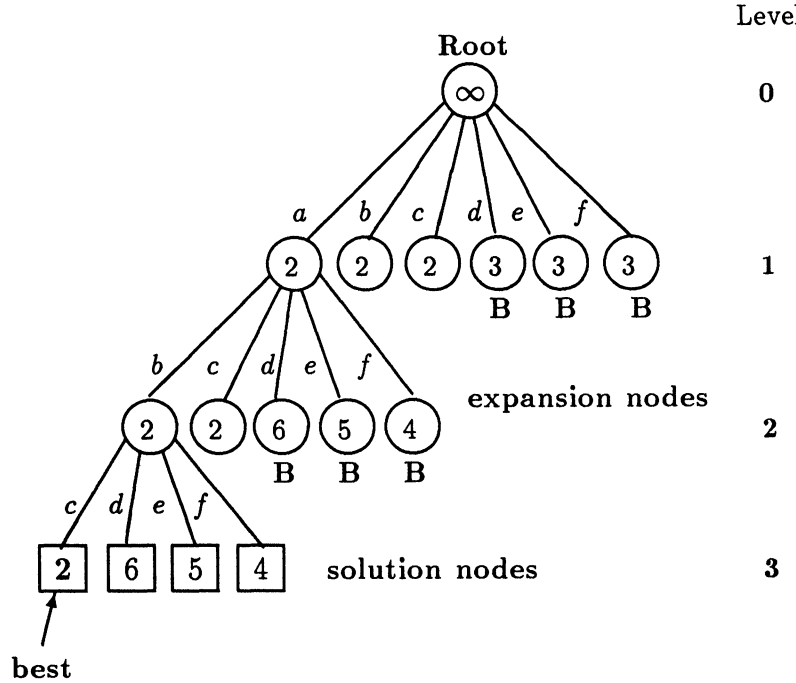


FIGURE 11 The search tree of $f = \bar{a}b(\bar{d}ef + df + \bar{e}\bar{f}) + \bar{b}e(\bar{e} + \bar{d}e + de\bar{f})$.

TABLE I
Comparisons of Branch and Bound Algorithm (B.B.) with Enumeration Method

	B.B.		Enumeration	
Circuits	*Sum*	*CPU*	*Sum*	*CPU*
5xp1	16	0.3	16	0.5
cm152a	7	0.4	7	0.5
cm162a	11	0.4	11	1.2
cm163a	11	0.3	11	0.6
cm85a	8	0.7	8	1.2
duke2	91	12.1	88	118.8
f51m	13	0.7	13	0.7
misex2	21	3.5	21	5.6
misex3c	96	105.5	94	2112.2
sao2	23	1.8	22	3.1
z4ml	6	0.4	6	0.5
cmp4	2	0.1	2	0.3
cmp6	2	1.4	2	4.4
cmp8	2	33.2	2	*
add4	2	0.8	2	4.0
add8	2	10.1	2	*

* indicates the CPU time exceeds two hours.

pansion nodes. In the search tree, each edge is tagged a variable $v \in X$. Each node N has a cost $Cube(C_f, P_N)$, where $P_N \subset X$ is the set of variables tagged on the path from **Root** to node N. The solution nodes are those nodes with $|P_N| = 3$.

Initially, the best cost *best* is initialized to ∞ and $\pi = (X_1 = \emptyset, X_2 = X)$. We find the best partition when $|X_1| = 1$, $X_1 = \{a\}$ is found to have the least cost. We now include the second variable into X_1 starting with $X_1 = \{a\}$. The second level expansion nodes from $X_1 = \{a\}$ are searched. The process continues until we find a solution, $X_1 = \{a, b, c\}$. Now we search for a better solution by expanding expansion nodes. Only those expansion nodes which have cost less than the *best* so far are searched. An expansion node with cost greater than the *best* is not searched because all solution nodes in this subtree have costs greater than the *best* by Observation 5.1. The expansion nodes with symbol **B** are bounded nodes. The search tree in Figure 11 shows that the optimal solution is the solution node with cost 2.

This estimated cost function has been tested for some circuits in the MCNC bench-marking set. Table I shows the results obtained by the estimated cost using branch and bound algorithm and by the exact cost using enumeration. The column *Sum* shows the sum of the number of equivalent classes of all outputs with respect to the partition found. The column *CPU* shows the running time in seconds. Note that example cmp$n$ is a n-bit comparator and add$n$ a n-bit adder. From the experimental results, our estimation cost can find the best partition for most cases.

The branch and bound algorithm is basically an exhaustive searching method. The worst case complexity is exponential in terms of input size. However, Table I shows that for large examples, our algorithm can significantly bound the search space.

5.3 Don't Care Assignment

In some cases, we are given an incompletely specified function. In the other cases, when deriving subfunctions, a don't care set maybe generated to f_t. This don't care set is very useful in searching a good decomposition. If the don't cares are assigned values properly, the number of equivalent classes can be minimized. We show an example to explain this.

Example 5.3 Given an incompletely specified function $f(a, b, c, d)$ and a partition $\pi = (\{a, b\}, \{c, d\})$. Its decomposition chart is shown in Figure 12(a). Two functions f_1 and f_2 are generated by different don't care assignments. Their decomposition charts are shown in Figure 12(b) and (c), respectively. We

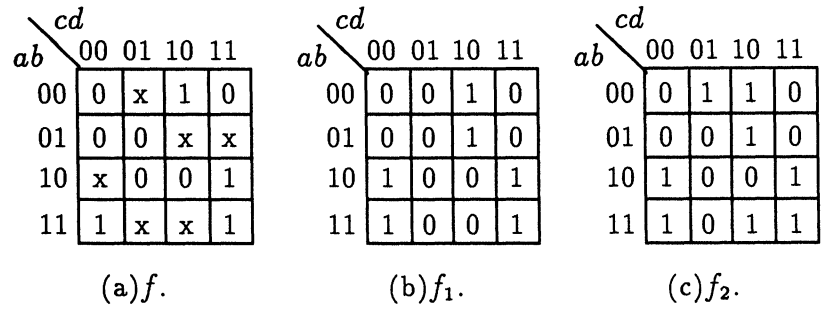


FIGURE 12 An example of don't care assignment.

can see that $EQU(f_1, \pi) = 2$ and $EQU(f_2, \pi) = 4$. □

Now a problem arises: "How to assign the don't cares to minimize the number of equivalent classes for incompletely specified functions?"

This problem has been formulated as the graph coloring problem in [25]. It uses a heuristic for coloring the graph with minimum colors. Each color is corresponding to an equivalent class. Rather than using graph coloring, a procedure is developed based on the algorithm proposed in [27]. This algorithm is used to compute the communication complexity for completely specified functions. We modify this algorithm to make it suitable for incompletely specified functions. We briefly review this algorithm below. The detailed procedure can be found in [34].

The algorithm used to compute the number of equivalent classes for incompletely specified function is mainly based on Lemma 1 in [31]. This lemma is repeated here as Lemma 5.1.

Lemma 5.1 Let $C_1(f)$ ($C_0(f)$) be the on (off)-set of $f(X)$ and $\pi = (X_1, X_2)$ is a partition. Given $v_0, v_1 \in BS(X_1)$, $v_0 \not\sim v_1$ if and only if there are cubes $(l_1, r_1) \in C_1(f)$ and $(l_0, r_0) \in C_0(f)$, where l_i and r_i are the coordinates associated with the variables in X_1 and variables in X_2, respectively, such that

(1) $r_0 \cap r_1 \neq \emptyset$

(2) $v_0 \in l_1$ and $v_1 \in l_0$, or $v_0 \in l_0$ and $v_1 \in l_1$. □

Our algorithm is mainly divided into two stages. The first stage is to partition all rows (elements in $BS(X_1)$) into many compatible groups considering the decomposition chart. In the same group, any two rows are mutually compatible. Two rows are compatible if the care elements are compatible; that is, for all vertically corresponding elements, it does not happen that one row is output 0 and the other is output 1. For example, rows 00 and 01 shown in Figure 12(a) are compatible; rows 00 and 10 are incompatible. Based on the above observation, we define an operator ★ shown in Table II(a). Lemma 5.2 is thus proposed for compatible checking.

TABLE II
Definitions of ★ and ⊎.

(a)

★	0	1	x
0	1	0	1
1	0	1	1
x	1	1	1

(b)

⊎	0	1	x
0	0	-	0
1	-	1	1
x	0	1	x

Lemma 5.2 Let $v_0, v_1 \in BS(X_1)$. Then $v_0 \sim v_1$ if and only if $f_{v_0} \bigstar f_{v_1} = 1$. □

The second stage is to assign the don't cares for each compatible groups. An operator ⊎ shown in Table II(b) is defined for don't care assignment. If two rows R_1 and R_2 are compatible, the rows after don't care assignment are $R_1 \uplus R_2$. We give an example to illustrate this.

Example 5.4 Consider the same function f and partition π shown in Example 5.3. By Lemma 5.2, rows 00 and 01 are compatible and thus form a compatible group G_1. A group G_2 of rows 10, 11, is formed in the same way. After don't care assignment, the rows for G_1 and G_2 are 0-0-1-0 and 1-0-0-1, respectively. The resultant decomposition chart is shown in Figure 12(b). □

Since this algorithm is divided into two stages, we analyze the time complexity for each stage. For the first stage, it is mainly based on the algorithm proposed in [27], so the time complexity is $O(pq)$, where p is the number of cubes in on-set of function f and q is the number of different overlappings of cubes, respectively. For the second stage, don't cares in the rows of a compatible group are assigned values according to the cubes of on-set and off-set in these rows. It is clear that the number of these cubes is no more than $p + r$, where p and r are the numbers of cubes in on-set and off-set of function f, respectively. The time complexity is therefore bounded by $O(m(p + r))$, where m is the number of compatible classes. By the above observations, the overall time complexity of this algorithm is $O(pq + m(p + r))$.

5.4 Encoding

The *Encoding* procedure is shown in Figure 13. It is mainly divided into two parts. The first part is to handle non-decomposable functions. In many cases, function $f(X)$ may be non-decomposable with respect to all partitions. In order to decompose f, a simple heuristic is proposed. The heuristic is to divide f into k subfunctions such that each subfunction has $\lceil m/k \rceil$ equivalent classes in $BS(X_1)$ with respect to π. Then f can be specified as:

$$f = f_1 + f_2 + \cdots + f_k. \tag{5}$$

Now f is feasible with respect to k and may be implemented by a single CLB. Then each function f_i is decomposed individually.

The second part deals with decomposable functions. The procedure includes deriving subfunctions

```
   Procedure Encoding(f, X, π)
1  if f is non-decomposable with respect to π then {
2      Divide f into k subfunctions f1, f2, ···, fk such that
3      f = f1 + f2 + ··· + fk; /* Now f = ft, fe = ∅ */
4      for i = 1 to k do {
           Fun-Map(fi(X));
       }
   } else {
5      Encode fe and deriving ft;
6      Generate don't care set to ft;
   }
7  return(ft, fe).
```

FIGURE 13 The *Encoding* procedure.

f_t and f_e. f_e is derived by assigning different binary codes to the equivalent classes. A heuristic that assignes two *similar* equivalent classes adjacent Gray-codes is proposed. *Similar* means their rows have the same output values in most of the corresponding vertically entries. For example, for the decomposition chart shown in Figure 14, there are 3 row patterns P_0, P_1 and P_2. P_0 and P_1 have 6 same outputs entries but P_1 and P_2 have only one same output. So we say that P_0 is more similar to P_1 than P_2. This heuristic is try to make f_t as simple as possible (with less number of cubes). After f_e is encoded, f_t is derived straightforward. In many cases, a don't care set may be generated to f_t. It happens when the number of equivalent classes is m and $2^{\lceil \log_2 m \rceil} > m$. This don't care set is very useful in the next level decomposition.

As shown in Figure 13, the *Encoding* procedure is divided into two parts. For nondecomposable functions, the dividing of f into k subfunctions f_i's can be done in constant time. So the time complexity of this part is $O(kT)$, where T is the time complexity of *Fun-Map*. For decomposable functions, the time complexity is dominated by the similarity checking among those equivalent classes. Consider two row patters P_i and P_j. The similarity between them depends on the on-set of $P_i \odot P_j$ ($\odot$ is an equivalence operation). That is, the larger the on set of $P_i \odot P_j$ the more similar they are. Using OBDD representation, this checking can be done in time complexity $O(S_iS_j)$, where S_i and S_j are the sizes of OBDD's representing P_i and P_j, respectively. Suppose there are m equivalent classes with row patterns P_1, P_2, . . . , P_m. The time complexity of similarity checking is then bounded by $O(m^2S^2)$, where $S = \max_i(S_i$: the size of OBDD representing $P_i)$.

X_1 \ X_2								
P_0	0	1	1	0	1	1	1	0
P_1	0	0	1	1	1	1	1	0
P_2	0	1	0	0	0	0	0	1

FIGURE 14 The example for similar rows.

6 EXPERIMENTAL RESULTS

The procedure, *Fun-Map,* has been implemented on the top of *MIS* [3] and runs on SUN4/370 (a 12.5 mips machine). The program reads in an input file written in *blif* format. The output networks are all verified by the "*verify*" command of *MIS* system.

We have tested examples from MCNC benchmarking set. In MCNC benchmarks, 9*symml, rd*84 and *rd*73 are totally symmetric functions. Table III gives the comparisons of *Fast-Decompose* and other tools in terms of area. Table IV shows the comparisons in terms of number of levels. The columns **C** and **L** represent the number of CLB's and the number of levels of the output network, respectively. Compared to the other systems, we have the best results in both area and delay.

Table V shows the results of *Fun-Map* compared with *TRADE* [25], *mispga* [22] and *mispga* (new) [23]. *TRADE* was also run on SUN4/370. Both of *mispga* and *mispga* (new) were run on the DEC 5500 (2.28 mips machine). In this table, column **T** represents the running time of the program measured by the *time* command of *MIS*. The experimental results show that our program runs better than *mispga* and *mispga* (new) in most of the benchmarks in terms

TABLE III
Comparisons of Number of CLBs

Circuits	*VISMAP* [12]	*chortle-crf* [16]	*Hydra* [19]	*Xmap* [20]	*MAPBDD* [21]	*mispga-91* [23]	*TRADE* [25]	*Fun-Map*
9symml	47	41	33	55	9	7	6	5
rd84	-	35	27	36	18	10	8	7
rd73	26	16	13	21	11	6	5	5

TABLE IV
Comparisons on Number of Levels

Circuits	*chortle-crf*		*chortle-d*		*mispga*		*XNFOPT* [13]		*TRADE*		*Fun-Map*	
	L	C	L	C	L	C	L	C	L	C	L	C
9symml	7	62	4	76	3	7	7	61	3	6	3	5
rd84	7	41	4	69	3	13	6	46	3	8	3	7
rd73	-	-	4	52	2	8	-	-	2	5	2	5

TABLE V
Comparisons Table

Ciruits	*Fun-Map*			*TRADE*			*mispga*			*mispga* (new)	
	C	L	T	C	L	T	C	L	T	C	T
alu2	93	6	26.6	22	3	12.2	122	6	42.6	109	773.8
9syml	5	3	5.1	6	3	4.9	7	3	15.2	7	339.7
9symml	5	3	1.3	6	3	4.9	7	3	9.9	7	127.2
rd73	5	2	1.0	5	2	3.7	8	2	4.4	6	24.0
rd84	7	3	3.4	8	3	11.6	13	3	9.8	10	73.7
f51m	10	3	0.8	9	3	2.3	23	4	5.9	17	14.4
5xp1	11	2	2.0	11	2	4.3	21	2	3.5	18	22.4
z4ml	4	2	0.3	4	2	2.0	10	2	2.1	5	5.0
sao2	22	3	6.7	27	3	13.8	45	5	9.5	28	41.9
bw	28	1	0.1	27	1	0.3	28	1	8.3	28	17.3
misex1	10	2	0.6	14	2	3.4	17	2	1.7	11	2.7
misex2	30	4	7.5	31	4	17.0	37	3	1.4	28	3.4
duke2	167	6	43.8	159	6	370.7	164	6	16.4	110	203.7
vg2	23	4	4.8				30			20	7.4
b9	32	4	11.2				42			39	27.6
cmb	14	4	0.9								
parity	11	4	30.5								
pcle	16	4	4.5								
pcler8	23	4	5.6								
pm1	14	3	5.8								

of both delay and area. Compared to *TRADE,* our program is also competitive. However, our program runs faster than *TRADE.*

7 DISCUSSION AND CONCLUSION

In this paper, functional decomposition approach is applied to RAM-based FPGA technology mapping. Using this approach, infeasible functions could be decomposed into several feasible subfunctions. Then each subfunction can be directly implemented by a CLB.

Our program is mainly divided into two parts. The first part is designed specifically for totally symmetric functions. The second part deals with general functions. In the first part, a weight based algorithm *Fast-Decompose* is proposed. In the second part, some methods such as output partition, variable partition, don't care assignment and encoding are considered to obtain better decomposition results. These techniques have been implemented and tested for many benchmarking examples. The results show that our algorithm is indeed effective. However, it is still possible to further improve the generated results. For example, the criterion used for handling non-decomposable functions can be improved. Currently, we are working toward finding a better technique for handling non-decomposable functions.

Acknowledgment

The authors would like to thank reviewers for their helpful suggestions.

References

[1] R.L. Ashenburst, "The decomposition of switching functions," *Ann. Comput. Lab. Harvard Univ.*

[2] R.E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Comput.,* Vol. C-35, No. 8. August 1986.

[3] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and Albert R. Wang, "MIS: a multiple-level logic optimization system," *IEEE Trans. on CAD,* Vol. 6, No. 6, pp. 1062–1081, Nov. 1987.

[4] H.A. Curtis, *A New Approach to the Design of Switching Circuits,* New York: Van Nostrand, 1962.

[5] K.C. Chen and S. Muroga, "Input assignment for decoded PLA's with multi-input decorders," in *ICCAD-88,* pp. 474–477, Nov. 1988.

[6] E. Detjens, et al., "Technology mapping in MIS," in *ICCAD-87,* pp. 116–119, Nov. 1987.

[7] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *DAC24,* pp. 341–347, June 1987.

[8] S. Devadas, Albert R. Wang, A.R. Newton and A. Sangiovanni-Vincentelli, "Boolean decomposition of programmable logic arrays," in *Proc. of Custom Integrated Circuit Conference,* May 1988.

[9] S. Devadas, Albert R. Wang, A.R. Newton and A. Sangiovanni-Vincentelli, "Boolean decomposition in multi-level logic optimization," in *ICCAD-88,* pp. 290–293, Nov. 1988.

[10] D.L. Dietmeyer and P.R. Schneider, "Identification of symmetry, redundancy and equivalence of Boolean functions," *IEEE Trans. Electron. Comput.,* Vol. 16, pp. 804–817, December 1967.

[11] B.G. Kim, and D.L. Dietmeyer, "Multilevel logic synthesis of symmetric switching functions," *IEEE Trans. on CAD,* Vol. 10, No. 4, pp. 436–446, April 1991.

[12] Nam-Sung Woo, "A heuristic method for FPGA technology mapping based on the edge visibility," in *DAC28,* pp. 248–251, June 1991.

[13] XACT LCA Development System, Vol. II, Xilinx Inc., 1989.

[14] S. Ercolani and G. De Micheli, "Technology mapping for electrically programmable gate arrays," in *DAC28,* pp. 234–239, June 1991.

[15] R.J. Francis, J. Roze, and K. Chung, "Chortle: A technology mapping for lookup table-based field programmable gate arrays," in *DAC27,* pp. 613–619, June 1990.

[16] R.J. Francis, J. Roze, and K. Chung, "Chortle-crf: Fast Technology mapping for lookup table-based FPGAs," in *DAC28,* pp. 227–233, June 1991.

[17] R.J. Francis, J. Roze, and K. Chung, "Technology mapping for lookup table-based FPGAs for performance," in *ICCAD-91,* pp. 568–571, Nov. 1991.

[18] R.J. Francis, J. Roze, and K. Chung, "Technology mapping for delay optimization of lookup table-based FPGAs," in *IWLS-91.*

[19] D. Filo, J.C. Yang, F. Mailhot, G. De Micheli, "Technology mapping for a two output RAM-based field programmable gate arrays," in *EDAC-91,* pp. 534–538, Feb. 1991.

[20] K. Karplus, "Xmap: a technology mapper for table-lookup field programmable gate arrays," in *DAC28,* June 1991.

[21] M.H. Tsai, T.T. Hwang, and Y.L. Lin, "Technology mapping for field programmable gate arrays using binary decision diagrams," *Synthesis and Simulation Meeting and International Interchange,* April 6–8, 1992.

[22] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in *DAC27,* pp. 620–625, June 1990.

[23] R. Murgai, N. Shenoy, R.K. Brayton and A. Sangiovanni-Vincentelli, "Improved logic synthesis algorithms for table lookup architecture," in *ICCAD-91,* pp. 564–567, Nov. 1991.

[24] R. Murgai, N. Shenoy, R.K. Brayton and A. Sangiovanni-Vincentelli, "Performance directed synthesis for table lookup programmable gate arrays," in *ICCAD-91,* pp. 572–575, Nov. 1991.

[25] Wei Wang and M.A. Perkowski, "A new approach to the decomposition of incompletely specified multiple output functions based on graph coloring and local transformations and its application to FPGA mapping," in *EURO-DAC'92,* pp. 230–235, June 1992.

[26] T.T. Hwang, R.M. Owens, and M.J. Irwin, "Exploiting communication complexity for multi-level logic synthesis," *IEEE Trans. on CAD,* Vol. 9, No. 10, pp. 1017–1027, October 1990.

[27] T.T. Hwang, R.M. Owens, and M.J. Irwin, "Efficiently computing communication complexity for multi-level logic synthesis," *IEEE Trans. on CAD,* Vol. 11, No. 5, pp. 545–554, May 1992.

[28] R.M. Karp, "Functional decomposition and switching circuit design," *J. Soc. Indus. Appl. Math.,* Vol. 11, No. 2, June 1963.

[29] E.J. McCluskey, *Logic Design Principles: with Emphasis on Testable Semicustom Circuit,* Prentice-Hall International Editions, 1986.

[30] S. Malik and R.H. Katz, "Combining multi-level decomposition and topological partitioning for PLAs," in *ICCAD-87,* pp. 112–115, Nov. 1987.

[31] J.P. Roth, R.M. Karp, "Minimization over boolean graphs," *IBM Journal,* pp. 227–238, April 1962.
[32] T. Sasao. "PLA decomposition," In *MCNC 1987 Logic Synthesis Workshop,* 1987.
[33] S. Yang, M.J. Giesielski, "PLA decomposition with generalized decoders," in *ICCAD-89,* pp. 312–315, Nov. 1989.
[34] K.H. Wang, "Study on functional decomposition for multilevel logic synthesis," Ph.D. dissertation of Department of Compute Science and Information Engineering, Chiao Tung University, Taiwan 30043, 1993.
[35] K.H. Wang, T.T. Hwang, C. Chen, "Restructure binary decision diagrams based on functional equivalence," in *EDAC'93,* pp. 261–265, Feb. 1993.
[36] Zvi, Kohavi, *Switching and Finite Automata Theory,* McGraw-Hill computer science series, 1987.

Biographies

Kuo-Hua Wang received the B.S. and M.S. degrees both in Computer Engineering from National Chiao Tung University, HsinChu, Taiwan, in 1986 and 1988, respectively. He is currently working toward the Ph.D. degree in the Department of Computer Science and Information Engineering, National Chiao Tung University.

His current research interests include logic synthesis and optimization, logic verification and high-level synthesis.

TingTing Hwang received the M.S. and Ph.D. degrees in computer science from the Pennsylvania State University in 1986 and 1990, respectively.

She is currently a faculty member of Computer Science with the National Tsing Hua University, Taiwan. Her research interests include multi-level logic synthesis and optimization, high-level synthesis.

Cheng Chen is a Professor in the Institute of Computer Science and Information Engineering, National Chiao Tung University, Taiwan, R.O.C. From 1972 to 1977, he was an instructor in the Department of Computer Science, NCTU. And from 1977 to 1981, he was an associate Professor in the Department of Computer Engineering. In the academic year 1980, he was a visiting scholar at the University of Illinois at Urbana-Champaign. Then he became full professor in the Department of Computer Engineering from 1981 until now. From 1987 to 1988, he was the chairman of the Department of Computer Engineering, National Chiao Tung University. In the academic year 1988, he was a visiting scholar at Carnegie-Mellon University. Currently, he is also the Deputy Director of Microelectronic and Information System Research Center, National Chiao Tung University. His research interests include computer architecture, parallel processing, compiling techniques for RISC and superscalar system, high performance inference machines.

Professor Chen received his M.S. degree from Institute of Electronics, National Chiao Tung University, in 1971. He is a member of IEEE Computer Society.