IEICE TRANS. FUNDAMENTALS, VOL.E90–A, NO.12 DECEMBER 2007

PAPER *Special Section on VLSI Design and CAD Algorithms*

BDD Representation for Incompletely Specified Multiple-Output Logic Functions and Its Applications to the Design of LUT Cascades

Munehiro MATSUURA† *and* **Tsutomu SASAO**†a), *Members*

SUMMARY A multiple-output function can be represented by a binary decision diagram for characteristic function (BDD_for_CF). This paper presents a method to represent multiple-output incompletely specified functions using BDD_for_CFs. An algorithm to reduce the widths of BDD_for_CFs is presented. This method is useful for decomposition of incompletely specified multiple-output functions. Experimental results for radix converters, adders, a multiplier, and lists of English words show that this method is useful for the synthesis of LUT cascades. An implementation of English words list by LUT cascades and an auxiliary memory is also shown.
key words: *incompletely specified function, characteristic function, binary decision diagram, functional decomposition, LUT cascade*

1. Introduction

Construction of a Binary Decision Diagram (BDD) for an incompletely specified Boolean function arises in several applications in the CAD domain: verification, logic synthesis, and software synthesis. Three methods are known to represent an incompletely specified logic function by binary decision diagrams (BDDs) [9]:

1. A ternary function that takes 0, 1 and *don't care* [9].
2. A pair of BDDs to represent three values [4].
3. An auxiliary variable that represents *don't cares* [3], [9].

Most works are related to the minimization of total number of nodes in BDDs [3], [6], [20], [21]. However, these methods are unsuitable for functional decompositions of multiple-output functions. In a functional decomposition, the minimization of width of a BDD is more important than the minimization of total number of nodes. To find an efficient decomposition of a multiple-output logic function, we can use a multi-terminal binary decision diagram (MTBDD), or a BDD that represents the characteristic function of the multiple-output function (BDD_for_CF) [15]. BDD_for_CFs usually require fewer nodes than corresponding MTBDDs, and the widths of the BDD_for_CFs tend to be smaller than that of the corresponding MTBDDs.

In this paper, we show a new method to represent an incompletely specified multiple-output function. It uses a BDD_for_CF, and is suitable for functional decomposition. We also show a method to reduce the width of the BDD_for_CF. Experimental results using radix converters, adders, a multiplier, and English word lists show the effectiveness of the approach. A preliminary version of this paper has been published as [17].

Manuscript received March 5, 2007.
Manuscript revised June 1, 2007.
Final manuscript received July 27, 2007.
†The authors are with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, 820-8502 Japan.
a) E-mail: sasao@cse.kyutech.ac.jp
DOI: 10.1093/ietfec/e90–a.12.2762

2. Definitions

Definition 2.1: x is a support variable of f if f depends on x. A function $f : \{0,1\}^n \rightarrow \{0,1,d\}$ is an incompletely specified function, where d denotes the *don't care*. Let f_0, f_1, and f_d be the functions represented by sets $f^{-1}(0)$, $f^{-1}(1)$, and $f^{-1}(d)$, respectively. Note that $f\_0 \vee f\_1 \vee f\_d = 1$, $f\_0 \cdot f\_1 = 0$, $f\_1 \cdot f\_d = 0$, and $f\_0 \cdot f\_d = 0$.

Definition 2.2: [2] Let $F = (f_1(X), f_2(X), \ldots, f_m(X))$ be a multiple-output function, and let $X = (x_1, x_2, \ldots, x_n)$ be the input variables. The characteristic function of the completely specified multiple-output function F is

$$\chi(X,Y) = \bigwedge_{i=1}^{m} (y_i \equiv f_i(X)),$$

where y_i is the variable representing the output f_i, and $i \in \{1,2,\ldots,m\}$.

The characteristic function of a completely specified multiple-output function denotes the set of the valid input-output combinations. Let $f_{i\_0}(X) = \bar{f_i}(X)$ and $f_{i\_1}(X) = f_i(X)$, then the characteristic function χ is represented as follows:

$$\chi(X,Y) = \bigwedge_{i=1}^{m} \{\bar{y}_i \cdot f_{i\_0}(X) \vee y_i \cdot f_{i\_1}(X)\}$$

In an incompletely specified function, when the function value f_i is *don't care*, the value of the function can be either 0 or 1. Therefore, for such inputs, the characteristic function is independent of the values of output variables y_i. Let f_{i_d} denote the *don't care* set, then we have

$$\begin{aligned} &\bar{y}_i(f_{i\_0}(X) \vee f_{i\_d}(X)) \vee y_i(f_{i\_1}(X) \vee f_{i\_d}(X)) \\ &= \bar{y}_i f_{i\_0}(X) \vee y_i f_{i\_1}(X) \vee f_{i\_d}(X) \end{aligned}$$

Thus, we have the following:

Definition 2.3: The characteristic function χ of an incompletely specified multiple-output function
$F = (f_1(X), f_2(X), \ldots, f_m(X))$ is

Table 1 Truth table of an incompletely specified function.

x_1	x_2	x_3	x_4	f_1	f_2	x_1	x_2	x_3	x_4	f_1	f_2
0	0	0	0	d	1	1	0	0	0	0	1
0	0	0	1	d	1	1	0	0	1	0	1
0	0	1	0	0	0	1	0	1	0	1	0
0	0	1	1	0	0	1	0	1	1	1	0
0	1	0	0	d	d	1	1	0	0	1	d
0	1	0	1	d	d	1	1	0	1	1	d
0	1	1	0	1	0	1	1	1	0	d	0
0	1	1	1	1	1	1	1	1	1	d	1

$$\chi(X,Y) = \bigwedge_{i=1}^{m} \{\bar{y}_i f_{i\_0}(X) \vee y_i f_{i\_1}(X) \vee f_{i\_d}(X)\}$$

Example 2.1: Consider the incompletely specified function shown in Table 1. Since,

$$\begin{aligned}
f_{1_0} &= \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2\bar{x}_3\\
f_{1_1} &= \bar{x}_1x_2x_3 \vee x_1\bar{x}_2x_3 \vee x_1x_2\bar{x}_3\\
f_{1_d} &= \bar{x}_1\bar{x}_3 \vee x_1x_2x_3\\
f_{2_0} &= \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2x_3 \vee x_2x_3\bar{x}_4\\
f_{2_1} &= \bar{x}_2\bar{x}_3 \vee x_2x_3x_4\\
f_{2_d} &= x_2\bar{x}_3,
\end{aligned}$$

the characteristic function is

$$\begin{aligned}
\chi = \{&\bar{y}_1(\bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2\bar{x}_3) \vee\\
&y_1(\bar{x}_1x_2x_3 \vee x_1\bar{x}_2x_3 \vee x_1x_2\bar{x}_3) \vee (\bar{x}_1\bar{x}_3 \vee x_1x_2x_3)\}\\
&\cdot\{\bar{y}_2(\bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2x_3 \vee x_2x_3\bar{x}_4) \vee\\
&y_2(\bar{x}_2\bar{x}_3 \vee x_2x_3x_4) \vee (x_2\bar{x}_3)\}
\end{aligned}$$

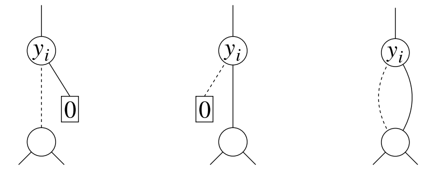
(End of Example)

Next, we will consider the BDD that represents the characteristic function for an incompletely specified multiple-output function.

Definition 2.4: The BDD_for_CF of a multiple-output function $F = (f_1, f_2, \ldots, f_m)$ represents the characteristic function χ of F, where the variable representing the output y_i is in the below of the support variables for f_i. (We assume that the root node is in the top.)

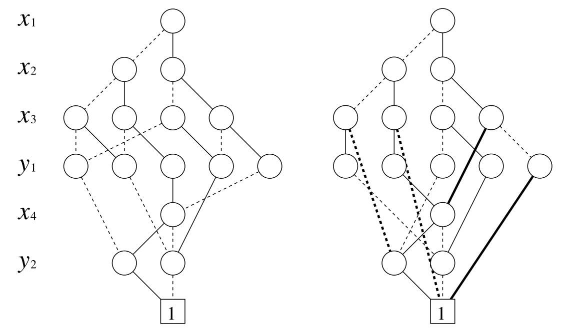
Figure 1 illustrates a BDD_for_CF of an incompletely specified multiple-output function, where solid lines denote the 1-edges, while dotted lines denote the 0-edges. When the 1-edge of the node y_i is connected to a constant 0 node, $f_i = 0$ (Fig. 1(a)); when the 0-edge of the node y_i is connected to a constant 0 node, $f_i = 1$ (Fig. 1(b)); and when both the 0-edge and 1-edge of the node y_i are connected to the same node except for the constant 0 node, $f_i = d$ (*don't care*) (Fig. 1(c)). In the case of $f_i = d$, the node for y_i is redundant, and it is deleted during the minimization of the BDD.

In the case of a BDD_for_CF representing a completely specified function, each path from the root node to the constant 1 node involves nodes for all the output variables y_i. Furthermore, one of the edges of the node for y_i is connected to the constant 0 node. On the other hand, in the case of a BDD_for_CF representing an incompletely specified function, each path from the root node to the constant 1 node may not involve nodes for some variable y_i. For the path where the output variables y_i is missing, f_i is *don't care*.



(a) $f_i = 0$ (b) $f_i = 1$ (c) $f_i = d$

Fig. 1 BDD_for_CF representing an incompletely specified function.



(a) BDD_for_CF when 0's are assigned to all the *don't cares*.
(b) BDD_for_CF for incompletely specified function.

Fig. 2 BDD_for_CF representing multiple-output function.

Example 2.2: Figure 2 shows two BDD_for_CFs representing the function in Example 2.1. For simplicity, the constant 0 node and all the edges connecting to it are omitted. Figure 2(a) shows the BDD_for_CF representing the completely specified function where 0's are assigned to all the *don't cares*. Figure 2(b) shows the BDD_for_CF representing the incompletely specified function. The solid and dotted bold edges denote that at least one node for output variables is missing, and the output value is *don't care*. Note that in Fig. 2(a), all the output variables $\{y_1, y_2\}$ appear in each path from the root node to the constant 1 node. On the other hand, in Fig. 2(b), in the bold edges at least one output variable y_i is missing, and the corresponding output f_i is *don't care*. (End of Example)

3. Decomposition and BDD_for_CFs

3.1 Decomposition Using BDD_for_CF

By using a BDD_for_CF, we can decompose a multiple-output logic function efficiently [15]. When we decompose the function by using a BDD, the smaller the width of the BDD, the smaller the network becomes after decomposition. In the case of an incompletely specified function, we can often reduce the width of the BDD_for_CF by finding an appropriate assignment of constants to the *don't cares*. From here, we will consider a method to reduce the width of a

Table 2 Decomposition chart of an incompletely specified function.

	$X_1 = \{x_1, x_2\}$			
$X_2 = \{x_3, x_4\}$	00	01	10	11
00	0	0	d	1
01	1	1	d	d
10	d	1	0	d
11	0	d	0	0
	Φ_1	Φ_2	Φ_3	Φ_4

BDD_for_CF representing an incompletely specified function.

Definition 3.5: Let the height of the root node be the total number of variables, and let the height of the constant node be 0. Let $(z_{n+m}, z_{n+m-1}, \ldots, z_1)$ be the ordering of the variables, where z_{n+m} corresponds to the variable for the root node. The width of the BDD_for_CF at the height k is the number of edges crossing the section of the BDD between variables z_k and z_{k+1}, where the edges incident to the same node are counted as one, also the edges pointing the constant 0 are not counted. The width of the BDD_for_CF at the height 0 is defined as 1.

Definition 3.6: Let $f(X)$ be a logic function, and (X_1, X_2) be a partition of the input variables. Let $|X|$ be the number of elements in X. The decomposition chart for f is a two-dimensional matrix with $2^{|X_1|}$ columns and $2^{|X_2|}$ rows, where each column and row has a label of unique binary code, and each element corresponds the truth value of f. In the decomposition chart, the column multiplicity denoted by μ is the number of different column patterns†. The function represented by a column pattern is a column function.

Example 3.3: Table 2 shows a decomposition chart of a 4-input 1-output incompletely specified function. Since all the column patterns are different, the column multiplicity is $\mu = 4$. (End of Example)

In a conventional functional decomposition using a BDD, the width of a BDD is equal to the column multiplicity μ. Let (X_1, X_2) be a partition of the input variables, then the nodes except for variables X_1 that are directly connected to the nodes for variables X_1 correspond to the column patterns in the decomposition chart. In a partition (X_1, X_2), nodes representing column functions may have different heights. In a functional decomposition, such a relation is denoted by $f(X_1, X_2) = g(h(X_1), X_2)$. When $\lceil \log_2 \mu \rceil < |X_1|$, the function f can be decomposed into two networks: The first one realizes $h(X_1)$, and the second one realizes $g(h, X_2)$. The functional decomposition is effective when the number of inputs for g is smaller than that of f.

Definition 3.7: Two incompletely specified functions f and g are compatible, denoted by $f \sim g$, iff $f\_0 \cdot g\_1 = 0$ and $f\_1 \cdot g\_0 = 0$.

Lemma 3.1: Let χ_a and χ_b be the characteristic functions of two incompletely specified functions. If $\chi_a \sim \chi_b$ and $\chi_c = \chi_a \chi_b$, then $\chi_c \sim \chi_a$ and $\chi_c \sim \chi_b$.

Example 3.4: In the decomposition chart of Table 2, for pairs of column functions $\{\Phi_1, \Phi_2\}$, $\{\Phi_1, \Phi_3\}$, and $\{\Phi_3, \Phi_4\}$, the functions are compatible. Make the logical product of columns Φ_1 and Φ_2, and replace them with Φ_1^* and Φ_2^*, respectively. Where, Φ_1^* and Φ_2^* show that the functions obtained by assigning constants to *don't cares*. Also, make the logical product of columns Φ_3 and Φ_4, and replace them with Φ_3^* and Φ_4^*, respectively. Then, we have the decomposition chart in Table 3, where $\mu = 2$.

(End of Example)

Table 3 Reduction of column multiplicity.

	$X_1 = \{x_1, x_2\}$			
$X_2 = \{x_3, x_4\}$	00	01	10	11
00	0	0	1	1
01	1	1	d	d
10	1	1	0	0
11	0	0	0	0
	Φ_1^*	Φ_2^*	Φ_3^*	Φ_4^*

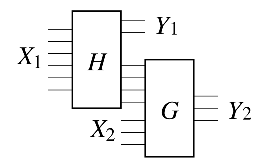


Fig. 3 Decomposition of multiple-output function.

The following theorem is similar to, but different from well known theorem on conventional functional decomposition using BDDs [8], [13]. It is an extension of [15] into incompletely specified functions.

Theorem 3.1: Let (X_1, Y_1, X_2, Y_2) be the variable ordering of the BDD_for_CF that represents the incompletely specified function, where X_1 and X_2 denote the disjoint ordered sets of input variables, and Y_1 and Y_2 denote the disjoint ordered sets of output variables. Let n_2 be the number of variables in X_2, and m_2 be the number of variables in Y_2. Let W be the width of the BDD_for_CF at height $n_2 + m_2$. When counting the width W, ignore the edges that connect the nodes of output variables and the constant 0. Suppose that the multiple-output function is realized by the network shown in Fig. 3. Then, the necessary and sufficient number of connections between two blocks H and G is $\lceil \log_2 W \rceil$.

3.2 Algorithm to Reduce the Width of a BDD_for_CF

Various methods exist to reduce the number of nodes in BDDs representing incompletely specified functions [3], [6], [21]–[23]. In the method [22], for each node, two children are merged when the functions represented by them are compatible. For example, when two children f and g in Fig. 4(a) are compatible, the BDD is simplified as shown in Fig. 4(b). By doing this operation repeatedly, we can reduce the number of nodes in the BDD. The following algorithm is used in our experiment, which is a simplified version of [22]. Note that our data structure is a BDD_for_CF instead of an SBDD.

†In the case of BDD_for_CF, we do not count the columns that consist of all zeros.

Algorithm 3.1: From the root node of the BDD, do the following operations recursively.

1. If the function represented by node v has no *don't care*, then terminate.
2. For v, check if two children v_0 and v_1 are compatible. Let the functions represented by v_0 and v_1 be χ_0 and χ_1, respectively.
    - If they are incompatible, then apply this algorithm to v_0 and v_1.
    - If they are compatible, then replace v_0 and v_1 with v_{new}, where v_{new} represents $\chi_{new}=\chi_0 \cdot \chi_1$, and apply this algorithm to v_{new}.

Example 3.5: Figures 5(a) and (b) show the BDD_for_CFs before and after the application of Algorithm 3.1, respectively. When there is only one edge coming down from a node, it denotes two edges that coincide. In Fig. 5(a), nodes 1 and 2 have compatible two children. For this function, the node replaced for node 1, and the node replaced for node 2 are the same. So, in Fig. 5(b), two nodes 1 and 2 are replaced with node 3. In the figures, the rightmost columns headed with "Width" denote the widths of the BDDs for each height. Note that the maximum width is reduced from 8 to 5, and the number of non-terminal nodes is reduced from 15 to 12. (End of Example)

The method in [22] is effective for local reduction of the number of nodes. However, since it only considers the compatibility of two children for each node at one time, it is not so effective to reduce the width of the BDD. Thus, in our method, we check the compatibility of column functions in each height, and perform the minimal clique cover of the functions to reduce the width of the BDD.

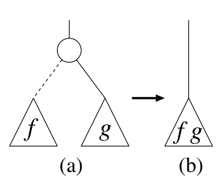


Fig. 4 Simplification method in [22].

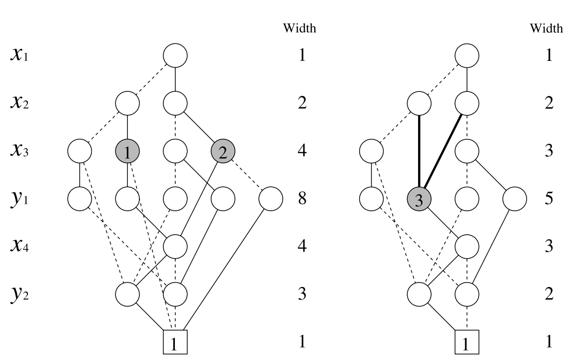


Fig. 5 BDD_for_CF before and after application of Algorithm 3.1.

Definition 3.8: In a compatibility graph, each node corresponds to a function, and an edge exists between nodes if the corresponding functions are compatible.

In the functional decomposition, check the compatibility of the column functions, and construct the compatibility graph. Then, minimize the column multiplicity μ by finding the minimum clique cover [3], [23]. Since this problem is NP-hard [5], we use the following heuristic method.

Algorithm 3.2: (Heuristic Minimal Clique Cover)
Let S_a be the set of all the nodes in the compatibility graph. Let C be the set of subset of S_a. From S_a, delete isolated nodes, and put them into C. While $S_a \neq \phi$, iterate the following operations:

1. Let v_i be the node that has the minimum number of edges in S_a. Let $S_i \leftarrow \{v_i\}$. Let S_b be the set of nodes in S_a that are connecting to v_i.
2. While $S_b \neq \phi$, iterate the following operations:
    a. Let v_j be the node that has the minimum edges in S_b. Let $S_i \leftarrow S_i \cup \{v_j\}$. $S_b \leftarrow S_b - \{v_j\}$.
    b. From S_b, delete the nodes that are not connected to v_j.
3. $C \leftarrow C \cup \{S_i\}$, $S_a \leftarrow S_a - S_i$.

Algorithm 3.3: (Reduction of Widths of a BDD_for_CF)
Let the height of the root node be t, and let the height of the constant nodes be 0. From the height $t-1$ to 1, iterate the following operations:

1. Construct the set of all the column functions, and construct the compatibility graph.
2. Find the minimum clique cover for the compatibility graph by Algorithm 3.2.
3. For each clique, make a function by AND operation of all functions that corresponds the nodes of the clique.
4. For each column function, replace it with the function produced in step 3, and re-construct the BDD with a smaller width.

Example 3.6: Figure 6 shows the BDD_for_CF after applying Algorithm 3.3 to Fig. 2(b). At the height of x_3, nodes 2 and 4 are compatible in Fig. 6(a). So, these nodes are merged into node **a** in Fig. 6(b). Next, at the height of y_1, the compatibility graph is shown in Fig. 7. Note that in Fig. 6(c), nodes 6 and 8 are replaced by node **b**, and nodes 7 and 10 are replaced by node **c**. The resulting BDD is shown in Fig. 6(d). By comparing Figs. 6(a) and (d), we can see that the maximum width is reduced from 8 to 4, and the number of non-terminal nodes is reduced from 15 to 12. (End of Example)

3.3 Reduction of Support Variables

In incompletely specified functions, some variables can be redundant [14]. In this case, such support variables can be removed by appropriately assigning values to the *don't cares*. Reduction of the support variables often reduces the

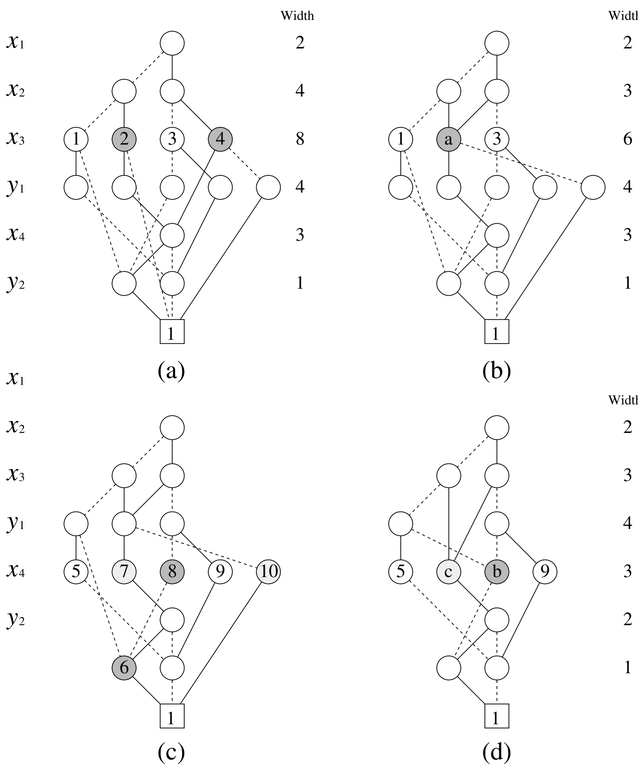


Fig. 6 Reduction of width of BDD_for_CF using Algorithm 3.3.

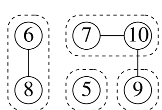


Fig. 7 Compatibility graph.

width of corresponding BDD_for_CF. Thus, we try to reduce support variables before applying Algorithm 3.1 or 3.3.

We use a greedy algorithm to reduce support variables. For each height in the BDD_for_CF, check whether the variable is redundant or not. Next, if it is redundant, then we delete the variable by appropriately assigning values to the *don't cares*. We apply the operation from the root to the leaf nodes.

4. Benchmark Functions

To evaluate the performance of Algorithm 3.3, we used the following incompletely specified functions.

4.1 Arithmetic Functions [17]

- Residue number to binary number converters.
- p-nary to binary converters.
- Decimal adders and multiplier.

Each of these functions represents a mapping $f : P_0 \times P_1 \times \cdots \times P_{k-1} \rightarrow Q$, where $P_i = \{0, 1, \ldots, p_i - 1\}$ and $Q = \{0, 1, \ldots, q-1\}$. In these benchmark functions, binary-coded-p_i-nary codes are used to represent p_i-nary digits. When p_i is not a power of 2, the p_i-nary digit has unused input combinations. In this case, we assign *don't cares* to the undefined outputs. We call such *don't cares* as input don't cares.

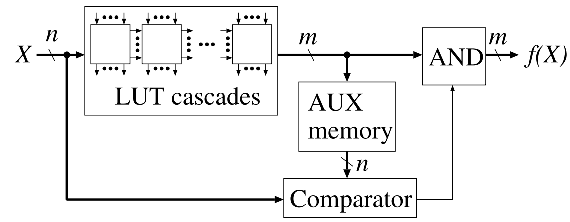


Fig. 8 Realization of English word list by using LUT cascade and AUX memory.

Let us consider the ratio of these *input don't cares* for benchmark functions. Note that each digit of a p_i-nary number uses $b_i = \lceil \log_2 p_i \rceil$ bits. Thus, the ratio of *input don't cares* is $\frac{2^{b_i} - p_i}{2^{b_i}} = 1 - \frac{p_i}{2^{b_i}}$, since each variable uses only p_i combinations out of 2^{b_i}. For a benchmark function with k digits, the ratio of *input don't cares* is

$$1 - \prod_{i=0}^{k-1} \frac{p_i}{2^{b_i}}.$$

Example 4.7: Consider the 10-digit ternary to binary converter. Assume that binary-coded-ternary is used to represent a ternary digit: 0 is represented by (00); 1 is represented by (01); and 2 is represented by (10). (11) is an undefined input, and the corresponding outputs are *don't cares*. Note that $p_0 = p_1 = p_2 = 3$, $b_0 = b_1 = b_2 = 2$, and $k = 10$. Thus, only $\left(\frac{3}{4}\right)^{10} = 0.0563$ of the input combinations are specified, and the remaining $1 - \left(\frac{3}{4}\right)^{10} = 0.9437$ of the combinations are unspecified. (End of Example)

4.2 English Word Lists [19]

We considered logic functions that represent three lists of English words [19] (Details are shown in Sect. 5.3.). For the English words consisting of fewer than 8 letters, we append blanks to the end of words to make them 8-letter words. Each English alphabet letter is represented by 5 bits, and each English word is represented by $n = 40$ bits. The numbers of words in the three lists are 1730, 3366, and 4705, respectively. In each word list, each English word has a unique index of an integer from 1 to k, where $k = 1730$ or 3366 or 4705. In this case, the outputs for undefined inputs are assigned to 0. The numbers of bits to represent the indices are $m = 11$, 12, and 13, respectively. Such a function denotes a mapping $f : P^8 \rightarrow Q$, where $P = \{0, 1, \ldots, 26\}$ and $Q = \{0, 1, \ldots, k\}$. In this case, the ratio of *input don't cares* is $1 - \left(\frac{27}{2^5}\right)^8 = 0.74$.

In the realization shown in Fig. 8 [19], we can replace the output 0 with *don't care*. In this case, the ratio of *don't cares* will be increased to $1 - \frac{k}{2^{40}}$. Note that for our benchmark function, only k different input combinations are mapped to integers from 1 to k, and other $(2^n - k)$ input combinations are mapped to *don't cares*. In Fig. 8, an English word list is implemented by using LUT cascades and an auxiliary memory [19]. The auxiliary memory checks whether output is correct or not. With this method, we can drastically reduce the size of the cascade.

5. Experimental Results

5.1 Reduction of BDD Width

We applied Algorithms 3.1 and 3.3 to each of the incompletely specified function presented in Sect. 4, and reduced the widths of the BDD_for_CF. Before applying algorithms, we optimized the order of the variables in the BDD_for_CF by sifting algorithm [12], where the sum of the widths is used as the cost function.

When all the output functions are represented by a single BDD_for_CF, the circuits were too large to implement. Handling many outputs at a time makes it difficult to find 0-1 assignments that simplify the BDD_for_CF. On the other hand, splitting outputs makes it easier to find 0-1 assignment to *don't cares*. However splitting all the outputs into single will conflict the optimization of multiple-output function. Similar things happen in the minimization of sum-of-products expressions for multiple-output functions. So, we partitioned the outputs into two sets, and represented each of them by a BDD_for_CF separately. Table 4 shows the maximum widths of the BDD_for_CF and the number of nodes when the multiple-output function $F = (f_1, \ldots, f_m)$ is partitioned into two: $F_1 = (f_1, \ldots, f_{\lceil m/2 \rceil})$ and $F_2 = (f_{\lceil m/2 \rceil + 1}, \ldots, f_m)$. The upper numbers denote the values for F_1, and the lower numbers denote the values for F_2.

In the table, the column headed by *In* denotes the number of inputs; *Out* denotes the number of outputs; *DC* denotes the ratio of *don't cares*; $DC = 0$ denotes the case where constant 0's were assigned to all the *don't cares*; $DC = 1$ denotes the case where constant 1's were assigned to all the *don't cares*; *ISF* denotes the case where incompletely specified functions (ternary functions) were represented; *Alg3.1* denotes the case where Algorithm 3.1 was applied; *Alg3.3* denotes the case where Algorithm 3.3 was applied; and *Time* denotes the computation time for Algorithms 3.1 and 3.3. Reduction ratio was normalized to 1.00 for the case of $DC = 0$.

By partitioning the outputs into two sets, we could drastically reduce the sizes of the BDDs for all the functions. For some functions, the maximum width of the BDD became less than 1/100 of the original BDD, and the total number of nodes became less than 1/30. With bi-partitions of outputs, we could implement the circuits with reasonable sizes.

Table 4 shows that Algorithm 3.3 produced BDDs with smaller widths than Algorithm 3.1, especially for F_2, the outputs for the least significant bits. Algorithm 3.1 checks only the compatibility of two children for each node, and reduces the number of nodes locally. On the other hand, Al-

Table 4 Maximum width and number of nodes in BDD_for_CF.

Function	In	Out	DC	Maximum width					# of nodes					Time [Sec]	
			[%]	DC=0	DC=1	ISF	Alg3.1	Alg3.3	DC=0	DC=1	ISF	Alg3.1	Alg3.3	Alg3.1	Alg3.3
5-7-11-13 RNS	14	13	69.5	426 375	426 375	426 375	425 374	395 320	2208 2022	2220 2022	2214 2028	1983 1742	1906 1741	0.011	0.156
7-11-13-17 RNS	16	15	74.0	450 897	450 897	450 897	449 896	316 896	2974 4730	2985 4730	2978 4737	2444 4254	2316 4073	0.033	0.562
11-13-15-17 RNS	17	16	72.2	1259 2143	1259 2144	1259 2144	1258 2143	777 1231	6830 9870	6845 9871	6838 9877	6271 9019	4576 7114	0.063	2.437
4-digit 11-nary to binary	16	14	77.7	117 256	117 256	117 257	116 256	115 128	1223 1931	1227 1931	1223 1935	1086 1582	1203 1328	0.017	0.094
4-digit 13-nary to binary	16	15	56.4	226 257	226 257	226 257	225 256	224 128	2293 2231	2299 2231	2293 2235	2150 1821	2288 1456	0.031	0.078
5-digit 10-nary to binary	20	17	90.5	393 257	393 257	393 78	392 76	391 64	3260 2322	3267 2322	3260 593	2794 527	3251 439	0.032	0.154
6-digit 5-nary to binary	18	14	94.0	134 257	134 257	134 257	133 256	129 128	1442 1875	1445 1875	1442 1878	1215 1373	1432 1337	0.030	0.063
6-digit 6-nary to binary	18	16	82.2	185 257	185 257	189 89	188 64	184 32	1310 1849	1317 1849	1367 445	1185 299	1328 307	0.032	0.015
6-digit 7-nary to binary	18	17	55.1	464 513	464 513	464 513	463 512	463 256	4917 4723	4923 4723	4917 4726	4566 3901	4826 3177	0.093	0.421
10-digit 3-nary to binary	20	16	94.4	265 513	265 513	265 513	264 512	240 256	2814 4005	2819 4005	2814 4007	2342 2842	2782 2961	0.063	0.328
3-digit decimal adder	24	16	94.0	27 200	27 101	14 14	13 13	10 10	187 1643	207 1035	129 125	93 95	109 108	0.046	0.000
4-digit decimal adder	32	20	97.7	79 1398	79 649	14 14	13 13	10 10	487 10047	509 5764	176 177	128 136	152 160	2.875	0.015
2-digit decimal multiplier	16	16	84.7	945 499	946 505	955 193	945 192	945 192	3013 2776	3020 2790	3035 1330	2716 1193	2980 1257	0.014	0.124
1730 words	40	11	99.9	866 834	901 836	735 843	383 427	100 149	16900 16616	17321 16670	9433 11478	1140 1698	954 2121	0.031	0.486
3366 words	40	12	99.9	1322 1709	1337 1713	1325 1729	634 975	192 441	17534 25434	17710 25446	14931 21510	1990 3428	2205 4729	0.063	1.704
4705 words	40	13	99.9	2025 2188	2042 2185	1895 2182	751 1161	213 385	33556 40627	33923 40506	27484 35159	2438 4561	2255 6027	0.078	2.673
Ratio				1.000	0.970	0.833	0.735	0.540	1.000	0.982	0.807	0.580	0.583		

Table 5 Reduction of LUT cascades by using *don't cares*.

Function	DC=0			Alg3.3		
	#Cel	#LUT	#Cas	#Cel	#LUT	#Cas
5-7-11-13 RNS	6	35	3	4	29	2
7-11-13-17 RNS	9	53	3	8	50	3
11-13-15-17 RNS	–	–	–	24	118	8
4-digit 11-nary to binary	4	29	2	4	28	2
4-digit 13-nary to binary	4	31	2	4	30	2
5-digit 10-nary to binary	8	50	3	6	48	2
6-digit 5-nary to binary	6	44	2	5	35	2
6-digit 6-nary to binary	5	33	2	4	29	2
6-digit 7-nary to binary	9	62	3	8	54	3
10-digit ternary to binary	9	58	3	6	48	2
3-digit decimal adder	6	35	2	3	17	1
4-digit decimal adder	9	62	2	4	24	1
2-digit decimal multiplier	8	51	3	7	48	3

gorithm 3.3 checks compatibilities among all the functions for each height of a BDD, to reduce the width more effectively. Let w be the width of the BDD, then the algorithm checks $(w^2 - w)/2$ compatibilities. Also, Algorithm 3.3 finds the minimal clique cover, so it is more time-consuming than Algorithm 3.1.

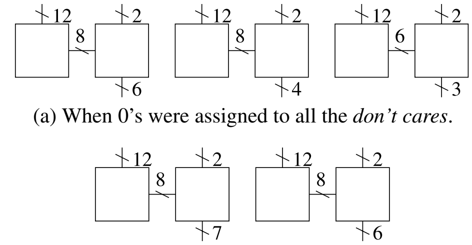
We used gcc version 3.2 compiler on a PC having Athlon64 (2.6 GHz) with 2 GByte memory. The longest CPU times were as follows: When all the outputs were represented by a single BDD_for_CF: 110 sec (the list of 4705 English words) to generate BDD_for_CF. 2433 sec (6-digit 7-nary to binary converter) to reduce the width of the BDD. When the outputs are bi-partitioned: 3.2 sec (11-13-15-17 RNS) to generate a pair of BDD_for_CFs. 2.4 sec (11-13-15-17 RNS) to reduce the widths of BDD_for_CFs.

5.2 Optimization of LUT Cascades

In the functional decomposition, when the width of the BDD is slightly larger than 2^k, by properly assigning the constants to *don't cares*, we can often reduce the width of the BDD, and reduces the number of interconnections between two blocks H and G in Fig. 3.

To see the usefulness of Algorithm 3.3, we designed benchmark functions in Sect. 4 by LUT cascades [15]. Table 5 shows the sizes of LUT cascades. For benchmark functions, we used cells with at most 12 inputs and at most 8 outputs to implement the cascades [11]. In Table 5, *#Cel* denote the number of cells in the cascade; *#LUT* denotes total number of LUT outputs; and *#Cas* denotes the number of cascades. The symbol '–' shows that the function could not be realized by LUT cascades.

For example, consider 5-7-11-13 RNS. In this function, 69.5% of the input combinations are *don't cares*. Figure 9(a) shows the case where constant 0's were assigned to all the *input don't cares*, while Fig. 9(b) shows the case where Algorithm 3.3 was used to assign *don't cares*. Algorithm 3.3 produced smaller cascades: On the average, the total numbers of cells is reduced by 22.4%, the total number of cell outputs is reduced by 17.9%, and the total numbers of cascades is reduced by 16.7%. We also designed cascades by using Algorithm 3.1. In this case, the reduction rates were, 16.4%, 14.6%, and 13.9%, respectively. So, Algorithm 3.1 was not so effective as Algorithm 3.3.



(a) When 0's were assigned to all the *don't cares*.

(b) When Algorithm 3.3 was used to assign *don't cares*.

Fig. 9 5-7-11-13 RNS to binary number converters.

Table 6 Realization of English word lists.

Design Method	# of words	#Cel	#LUT	#Cas	#RV	MemBits	
						LUT	AUX
DC=0	1730	26	237	2	0	954,624	0
	3366	60	475	6	0	1,892,416	0
	4705	132	1094	12	0	4,279,936	0
Fig. 8	1730	5	36	1	9	110,592	81,920
	3366	11	77	2	9	258,048	163,840
	4705	14	100	2	3	310,272	327,680

5.3 Realization of English Word Lists by LUT Cascade and Auxiliary Memory

We designed English word lists by the architecture in Fig. 8. Although this architecture requires the auxiliary memory with $n2^m$ bits, we can drastically reduce the number of cells and number of memory bits for LUT cascades. Hence, we have a faster circuit. By replacing the output 0 with *don't care*, we often have a function with redundant variables. If the cascade consists of a single memory, reduction of i variables reduces the size of memory into $\frac{1}{2^i}$.

Table 6 shows the size of LUT cascades and auxiliary memory. In Table 6, $DC = 0$ denotes the case where circuits were designed by only LUT cascades; *Fig. 8* denotes the case where circuits were designed by architecture in Fig. 8; *#Cel* denote the number of cells in the cascade; *#LUT* denotes total number of LUT outputs; *#Cas* denotes the number of cascades; *#RV* denotes the number of redundant variables; *LUT* in *MemBits* denotes the total memory bits for LUT cascades; and *AUX* in *MemBits* denotes the memory bits for a auxiliary memory. To implement the cascade, the case of *DC=0* requires 12-input 10-output cells. Thus, we used cells with at most 12 inputs and at most 10 outputs.

Table 6 shows that the total number of LUT outputs is reduced by 83.8% to 90.9%; the total number of cells is reduced by 80.8% to 89.4%; the number of memory bits for LUT cascades is reduced by 86.4% to 92.8%; and the number of memory bits including the auxiliary memory is reduced by 77.7% to 85.1%. We could remove 9 variables for the lists of 1730 word and 3366 word, and could remove 3 variables for the lists of 4705 word.

6. Concluding Remarks

In this paper, we first showed a new method to represent an incompletely specified multiple-output function by a BDD_for_CF. Second, we presented a method to reduce the width of the BDD. Third, we applied this method to radix converters, adders, a multiplier, and lists of English words. When all the outputs were represented by a single BDD_for_CF, we could not reduce the width of the BDD even if we use the *don't cares*. However, when the outputs were partitioned into two sets, and each set was represented by a BDD_for_CF, we could reduce the width of the BDD by using *don't cares*. We also applied this method to design LUT cascades. By using the method, we could reduce the numbers of cells in cascades, on the average, by 22.4%.

Acknowledgments

This research is supported in part by Grant in Aid for Scientific Research of MEXT, and Grant of Kitakyushu Area Innovative Cluster Project.

References

[1] R.L. Ashenhurst, "The decomposition of switching functions," International Symposium on the Theory of Switching, pp.74–116, April 1957.
[2] P. Ashar and S. Malik, "Fast functional simulation using branching programs," Int. Conf. on CAD, pp.408–412, Nov. 1995.
[3] S. Chang, D. Cheng, and M. Marek-Sadowska, "Minimizing ROBDD size of incompletely specified multiple output functions," European Design & Test Conf., pp.620–624, 1994.
[4] K. Cho and R.E. Bryant, "Test pattern generation for sequential MOS circuits by symbolic fault simulation," Design Automation Conference, pp.418–423, June 1989.
[5] M.R. Garey and D.S. Johnson, Computers and Intractability, Freeman, San Francisco, 1979.
[6] Y. Hong, P. Beerel, J. Burch, and K. McMillan, "Safe BDD minimization using don't cares," Design Automation Conference, pp.208–213, 1997.
[7] I. Koren, Computer Arithmetic Algorithms, 2nd ed., A.K. Peters, Natick, MA, 2002.
[8] Y.-T. Lai, M. Pedram, and S.B.K. Vrudhula, "BDD based decomposition of logic functions with application to FPGA synthesis," Design Automation Conference, pp.642–647, 1993.
[9] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," Design Automation Conference, pp.52–57, June 1990.
[10] A. Mishchenko and T. Sasao, "Encoding of Boolean functions and its application to LUT cascade synthesis," International Workshop on Logic and Synthesis 2002, pp.115–120, New Orleans, Louisiana, June 2002.
[11] K. Nakamura, T. Sasao, M. Matsuura, K. Tanaka, K. Yoshizumi, H. Qin, and Y. Iguchi, "Programmable logic device with an 8-stage cascade of 64K-bit asynchronous SRAMs," Cool Chips VIII, IEEE Symposium on Low-Power and High-Speed Chips, pp.382–389, April 2005.
[12] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," International Conference on Computer Aided Design, pp.42–47, Nov. 1993.
[13] T. Sasao, "FPGA design by generalized functional decomposition," in Logic Synthesis and Optimization, pp.233–258, Kluwer Academic Publisher, 1993.
[14] T. Sasao, "On the number of dependent variables for incompletely specified multiple-valued functions," International Symposium on Multiple-Valued Logic, pp.91–97, May 2000.
[15] T. Sasao and M. Matsuura, "A method to decompose multiple-output logic functions," Design Automation Conference, pp.428–433, San Diego, June 2004.
[16] T. Sasao, "Radix converters: Complexity and implementation by LUT cascades," International Symposium on Multiple-Valued Logic, pp.256–263, May 2005.
[17] T. Sasao and M. Matsuura, "BDD representation for incompletely specified multiple-output logic functions and its applications to functional decomposition," Design Automation Conference, pp.373–378, June 2005.
[18] Available at http://www.lsi-cad.com/dac2005
[19] T. Sasao, "A design method of address generators using hash memories," International Workshop on Logic and Synthesis, pp.102–109, June 2006.
[20] M. Sauerhoff and I. Wegener, "On the complexity of minimizing the OBDD size for incompletely specified functions," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.15, no.11, pp.1435–1437, 1996.
[21] C. Scholl, "Multi-output functional decomposition with exploitation of don't cares," Design Automation and Test Europe, pp.743–748, Feb. 1998.
[22] T.R. Shiple, R. Hojati, A.L. Sangiovanni-Vincentelli, and R.K. Brayton, "Heuristic minimization of BDDs using don't cares," Design Automation Conference, pp.225–231, 1994.
[23] W. Wan and M.A. Perkowski, "A new approach to the decomposition of incompletely specified functions based on graph-coloring and local transformations and its application to FPGA mapping," IEEE EURO-DAC'92, pp.230–235, Hamburg, Sept. 1992.

Munehiro Matsuura was born on 1965 in Kitakyushu City, Japan. He studied at the Kyushu Institute of Technology from 1983 to 1989. He received the B.E. degree in Natural Sciences from the University of the Air, in Japan, 2003. He has been working as a Technical Assistant at the Kyushu Institute of Technology since 1991. He has implemented several logic design algorithms under the direction of Professor Tsutomu Sasao. His interests include decision diagrams and exclusive-OR based circuit design.

Tsutomu Sasao received the B.E., M.E., and Ph.D. degrees in Electronics Engineering from Osaka University, Osaka Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan, IBM T. J. Watson Research Center, Yorktown Height, NY and the Naval Postgraduate School, Monterey, CA. He has served as the Director of the Center for Microelectronics Systems at the Kyushu Institute of Technology, Iizuka, Japan. Now, he is a Professor of Department of Computer Science and Electronics. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than 8 books on logic design including, *Logic Synthesis and Optimization, Representation of Discrete Functions, Switching Theory for Logic Synthesis, and Logic Synthesis and Verification*, Kluwer Academic Publishers 1993, 1996, 1999, 2002 respectively. He has served Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan in 1998. He received the NIWA Memorial Award in 1979, Takeda Techno-Entrepreneurship Award in 2001, and Distinctive Contribution Awards from IEEE Computer Society MVL-TC for papers presented at ISMVLs, in 1987, 1996, 2003. He has served an associate editor of the *IEEE Transactions on Computers*. He is a Fellow of the IEEE.