BDD Representation for Incompletely Specified Multiple-Output Logic Functions and Its Applications to Functional Decomposition

Tsutomu Sasao and Munehiro Matsuura
Department of Computer Science and Electronics,
Kyushu Institute of Technology,
Iizuka 820-8502, Japan

ABSTRACT

A multiple-output function can be represented by a binary decision diagram for characteristic function (BDD_for_CF). This paper presents a new method to represent multiple-output incompletely specified functions using BDD_for_CF. 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 and a multiplier show that this method is useful for the synthesis of LUT cascades. This data structure is also useful to three-valued logic simulation.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids

General Terms

Algorithms, Design, Experimentation, Performance, Theory

Keywords

Incompletely Specified Function, BDD, Characteristic function, Cascade, Code converter

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* [9, 3].

Most works are related to the minimization of total number of nodes in BDDs [3, 6, 17, 18]. 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) [14]. 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, and a multiplier show the effectiveness of the approach.

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

2. DEFINITIONS

DEFINITION 2.1. *x is a* **support variable** *of f if f depends on x. A function $f : \{0,1\}^n \to \{0,1,d\}$ 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 $i \in \{0,1, \ldots, m\}$.

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

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

Table 2.1: 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

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*

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

EXAMPLE 2.1. *Consider the incompletely specified function shown in Table 2.1. 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

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

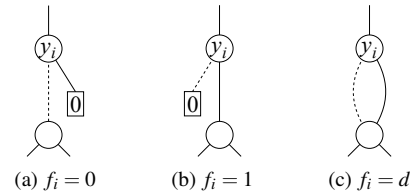
∎

Next, we will consider the BDD that represents the characteristic function for 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.)*

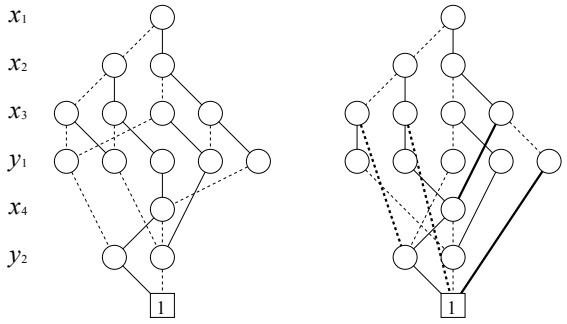
Fig. 2.1 illustrates a BDD_for_CF of an incompletely specified multiple-output function, where solid lines denote the 1-edges, while dotted lines denote the 0-edges. When the 1-edge of the node y_i is connected to a constant 0 node, $f_i = 0$ (Fig. 2.1(a)); when the 0-edge of the node y_i is connected to a constant 0 node, $f_i = 1$ (Fig. 2.1(b)); and when both the 0-edge and 1-edge of the node y_i are connected to the same non-constant 0 node, $f_i = d$ (*don't care*) (Fig. 2.1(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) $f_i = d$

Figure 2.1: BDD_for_CF representing an incompletely specified function.



(a) BDD_for_CF when 0's are assigned to the all the *don't cares*.

(b) BDD_for_CF for incompletely specified function.

Figure 2.2: BDD_for_CF representing multiple-output function.

EXAMPLE 2.2. *Fig. 2.2 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. Fig. 2.2(a) shows the BDD_for_CF representing the completely specified function, where* 0*'s are assigned to all the don't cares. Fig. 2.2(b) shows the BDD_for_CF representing the incompletely specified function. The solid and dotted bold edges denote that at least one node for an output variable is missing, and the output value is don't care. Note that in Fig. 2.2(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. 2.2(b), in the bold edges at least one output variable y_i is missing, and the corresponding output f_i is don't care.* ∎

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 [14]. 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 a 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 BDD_for_CF representing an incompletely specified function.

DEFINITION 3.1. *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*

Table 3.1: Decomposition chart of an incompletely specified function.

$X_2=\{x_3,x_4\}$ \ $X_1=\{x_1,x_2\}$	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

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 3.2. *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* [1]*. The function represented by a column is a* **column function***.*

EXAMPLE 3.1. *Table 3.1 shows a decomposition chart of a 4-input 1-output incompletely specified function. Since all the column patterns are different, the column multiplicity is $\mu = 4$.* ∎

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 3.3. *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 3.2. *In the decomposition chart of Table 3.1, 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 3.2, where $\mu = 2$.* ∎

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 [14] into incompletely specified functions.

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

Table 3.2: Reduction of column multiplicity.

$X_2=\{x_3,x_4\}$ \ $X_1=\{x_1,x_2\}$	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^*

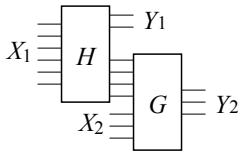


Figure 3.1: Decomposition of multiple-output function.

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. 3.1. 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, 18, 19, 20]. In the method [19], 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. 3.2(a) are compatible, the BDD is simplified as shown in Fig. 3.2(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 [19]. Note that our data structure is a BDD_for_CF instead of an SBDD.

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.*

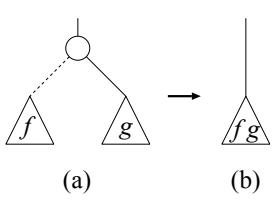
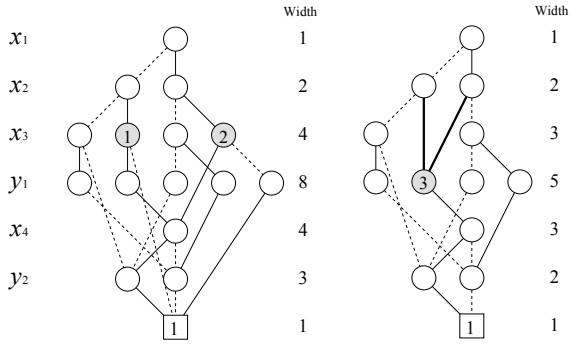


Figure 3.2: Simplification method in [19].



(a) Before application. (b) After application.

Figure 3.3: BDD_for_CF before and after application of Algorithm 3.1.

- *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 $\chi_{new}=\chi_0 \cdot \chi_1$.*

EXAMPLE 3.3. *Fig. 3.3(a) and (b) show the BDD_for_CFs before and after 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. 3.3(a), nodes* 1 *and* 2 *have compatible two children. For this function, the node replaced for node 1, and the node replaced for nodes 2 are the same. So, in Fig. 3.3(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*.* ∎

The method in [19] 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.

DEFINITION 3.4. *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 [20, 3]. 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. *$C \leftarrow C \cup \{S_i\}$, $S_a \leftarrow S_a - S_i$.*

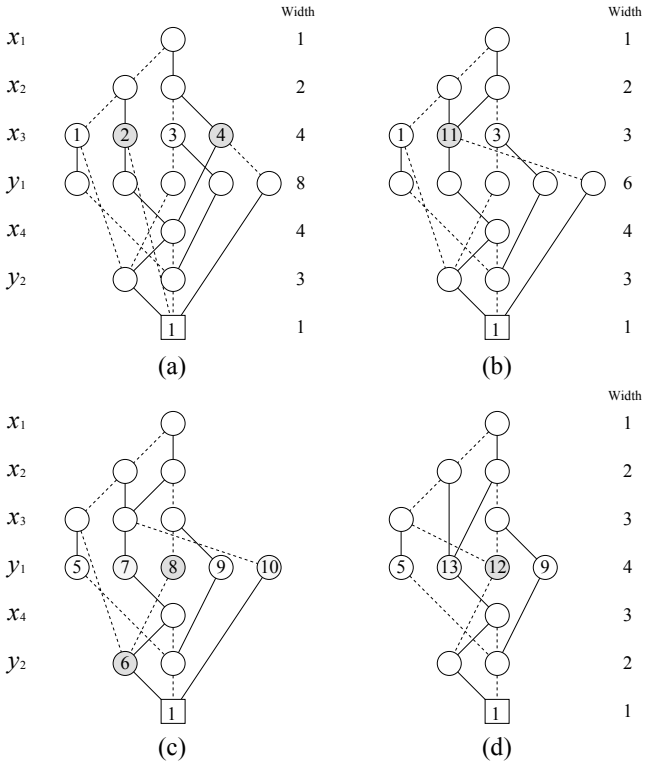


Figure 3.4: Reduction of width of BDD_for_CF using Algorithm 3.3.

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. *For each height, construct the set of all the column functions, and construct the compatibility graph.*
2. *Find the minimum clique cover for the nodes by Algorithm 3.2.*
3. *For the functions that corresponds the nodes of each clique, assign constants to the don't cares to make a function that is compatible to all the function.*
4. *For each column function, replace it with the function produced in step* 3*, and re-construct the BDD with a smaller width.*

EXAMPLE 3.4. *Fig. 3.4 shows the BDD_for_CF after applying Algorithm 3.3 to one in Fig. 2.2. At the height of x_3, nodes* 2 *and* 4 *are compatible in Fig. 3.4(a). So, these nodes are merged into node* 11 *in Fig. 3.4(b). Next, at the height of y_1, the compatibility graph is shown in Fig. 3.5. Note that in Fig. 3.4(c), nodes* 6 *and* 8 *are replaced by node* 12*, and nodes* 7 *and* 10 *are replaced by node* 13*. The resulting BDD is shown in Fig. 3.4(d). By comparing Figs. 3.4(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*.* ∎

4. EXPERIMENTAL RESULTS

4.1 Benchmark Functions

To evaluate the performance of Algorithm 3.3, we generated the following incompletely specified functions [16, 15]:

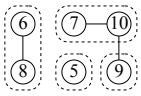


Figure 3.5: Compatibility graph.

Residue Number to Binary Number Converters:

- 5-7-11-13 RNS (14-input 13-output, DC 69.5%: 4-digit RNS number [7], where the moduli are 5, 7, 9, and 11)
- 7-11-13-17 RNS (16-input 15-output, DC 74.0%)
- 11-13-15-17 RNS (17-input 16-output, DC 72.2%)

k-nary to Binary Converters:

- 4-digit 11-nary to binary (16-input 14-output, DC 77.7%)
- 4-digit 13-nary to binary (16-input 15-output, DC 56.4%)
- 5-digit decimal to binary (20-input 17-output, DC 90.5%)
- 6-digit 5-nary to binary (18-input 14-output, DC 94.0%)
- 6-digit 6-nary to binary (18-input 16-output, DC 82.2%)
- 6-digit 7-nary to binary (18-input 17-output, DC 55.1%)
- 10-digit ternary to binary (20-input 16-output, DC 94.4%)

Decimal Adders and a Multiplier:

- 3-digit decimal adder (24-input 16-output, DC 94.0%)
- 4-digit decimal adder (32-input 20-output, DC 97.7%)
- 2-digit decimal multiplier (16-input 16-output, DC 84.7%)

EXAMPLE 4.1. *Consider the 10-digit ternary to binary converters. 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. Thus,* $(\frac{3}{4})^{10} = 0.0563$ *of the inputs are specified, but* $1-(\frac{3}{4})^{10} = 0.9436$ *are don't cares.* ∎

4.2 Reduction of BDD Width

We applied Algorithms 3.1 and 3.3 to each of the incompletely specified function presented in Section 4.1, and reduced the widths of the BDD_for_CF. Before applying Algorithms, we optimize the variable order for the BDD_for_CF by sifting algorithm [12], where the sum of widths is used as the cost function.

When all the outputs of the function are represented by a single BDD_for_CF, the circuits are too large to implement. So, we partition the outputs into two sets, and represent each of them by a BDD_for_CF separately. Table 4.1 shows maximum widths of BDD_for_CF 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 results for F_1, and the lower numbers denote the results for F_2. In the tables, the column headed by $DC = 0$ denotes the case where constant 0's are assigned to all the *don't cares*; $DC = 1$ denotes the case where constant 1's are assigned to all the *don't cares*; ISF denotes the case where incompletely specified functions (ternary functions) are represented; Alg3.1 denotes the case where Algorithm 3.1 is applied; and Alg3.3 denotes the case where Algorithm 3.3 is applied. Reduction ratio is 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 function, the maximum width of the BDD became $1/2000$, and the total number of nodes became $1/70$. With bi-partitions, we can implement the circuits with reasonable sizes.

Table 4.1 shows that Algorithm 3.3 produced BDDs with smaller widths than Algorithm 3.1, especially for F_2, the outputs for less 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, Algorithm 3.3 checks compatibilities among functions for each height of a BDD, and so reduces the width more effectively. Let w be the width of the BDD, then the algorithm checks $(w^2 - w)/2$ compatibilities. Also, it performs minimal clique cover, so it is more time-consuming than Algorithm 3.1.

We used gcc for Red Hat Linux version 2.4 on a PC using Pentium 4 (2.8GHz) with 4G Byte memory. The longest CPU times were as follows: When all the outputs were represented by a single BDD_for_CF: 135 sec (7-11-13-17 RNS) to generate BDD_for_CF. 783 sec (10-digit ternary to binary) to reduce the width of the BDD. When the outputs are bi-partitioned: 26 sec (11-13-15-17 RNS) to generate a pair of BDD_for_CFs. 11 sec (11-13-15-17 RNS) to reduce the widths of BDD_for_CFs.

Table 4.1: Maximum width of BDD_for_CFs.

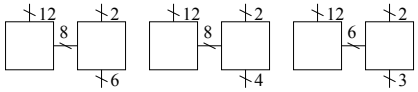
Name	DC=0	DC=1	ISF	Alg3.1	Alg3.3
5-7-11-13 RNS	503 461	503 461	503 461	502 460	416 363
7-11-13-17 RNS	471 1089	471 1089	471 1089	470 1088	337 896
11-13-15-17 RNS	1574 2253	1574 2253	1574 2254	1573 2253	1573 1229
4-digit 11-nary to binary	117 257	117 257	129 264	123 264	117 128
4-digit 13-nary to binary	226 257	226 257	236 264	232 264	225 128
5-digit decimal to binary	393 257	393 257	393 78	392 76	391 64
6-digit 5-nary to binary	134 257	134 257	155 260	148 260	139 128
6-digit 6-nary to binary	185 257	185 257	189 89	188 64	184 32
6-digit 7-nary to binary	464 513	464 513	485 516	482 516	472 256
10-digit ternary to binary	265 513	265 513	305 514	304 514	294 256
3-digit decimal adder	29 200	25 101	15 14	14 13	10 10
4-digit decimal adder	85 1400	73 649	15 14	14 13	10 10
2-digit decimal multiplier	945 767	946 769	955 327	954 269	945 224
Reduction ratio	1.000	0.950	0.825	0.811	0.636

Table 4.2: Reduction of LUT Cascades by using *don't cares*.

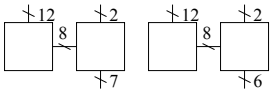
Name	DC=0			Alg3.3		
	#Cells	#LUTs	#Cas	#Cells	#LUTs	#Cas
5-7-11-13 RNS	6	35	3	4	29	2
7-11-13-17 RNS	9	53	3	8	52	3
11-13-15-17 RNS	19	118	5	18	108	5
4-digit 11-nary to binary	4	29	2	4	28	2
4-digit 13-nary to binary	4	31	2	4	30	2
5-digit decimal to binary	8	50	3	7	48	2
6-digit 5-nary to binary	6	44	2	5	36	2
6-digit 6-nary to binary	5	38	2	5	29	2
6-digit 7-nary to binary	10	68	3	9	58	3
10-digit ternary to binary	9	59	3	6	48	2
3-digit decimal adder	7	37	2	3	17	1
4-digit decimal adder	10	64	2	4	24	1
2-digit decimal multiplier	8	51	3	7	48	3
Total	105	677	35	84	555	30
Ratio	1.00	1.00	1.00	0.80	0.82	0.86

4.3 Logic Synthesis of LUT Cascades

In the practical applications, 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 re-



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



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

Figure 4.1: 5-7-11-13 RNS to binary number converters.

duces the number of interconnections between two blocks H and G in Fig. 3.1.

To see the usefulness of Algorithm 3.3, we designed benchmark functions in Section 4.1 by LUT cascades [14] . Table 4.2 shows the sizes of LUT cascades. For all benchmark functions except for 11-13-15-17 RNS, we assume that each cell has at most 12 inputs and at most 8 outputs [11]. For 11-13-15-17 RNS, we used cells with at most 12 inputs and at most 9 outputs, since it required 9-output cells. In the table, #Cells denote the number of cells in the cascade; #LUTs denotes total number of LUT outputs; and #Cas denotes the number of cascades.

For example, consider 5-7-11-13 RNS. In this function, 69.5% of the input combinations are *don't cares*. Fig. 4.1(a) shows the case where constant 0's were assigned to all the *don't cares*, while Fig. 4.1(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 by 20%, the total number of cell outputs is reduced by 18%, and the total numbers of cascades is reduced by 14%.

We also designed cascades by using Algorithm 3.1. In this case, the reduction were, 15%, 14%, and 11%, respectively. So, it was not so effective as Algorithm 3.3.

5. CONCLUDING REMARKS

In this paper, we first showed a new method to represent an incompletely specified multiple-output function by a BDD_for_CF. Then, we presented a method to reduce the width of the BDD. Finally, we applied this method to radix converters, adders and a multiplier. 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 20%.

The technique is promising for other BDD-based synthesis areas, e.g. BDD-based state-space exploration, where the BDD representation of the state set can be simplified using reached states as *don't cares*.

Acknowledgments

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

6. 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," *Inter. 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," *In 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 Edition)*, 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*, 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 4-7, 2002.

[11] K. Nakamura, and T. Sasao, *et.al* "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, April 20-22, 2005, Yokohama, Japan (to be published).

[12] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *ICCAD-93*, pp. 42–47, 1993.

[13] T. Sasao, "FPGA design by generalized functional decomposition," In *Logic Synthesis and Optimization*, Kluwer Academic Publisher, pages 233–258, 1993.

[14] T. Sasao and M. Matsuura, "A method to decompose multiple-output logic functions," *Design Automation Conference*, pp. 428-433, San Diego, June 2-6, 2004.

[15] T. Sasao, "Radix converters: Complexity and implementation by LUT cascades," *International Symposium on Multiple-Valued Logic*, May 2005 (to be published).

[16] Available at http://www.lsi-cad.com/dac2005

[17] M. Sauerhoff and I. Wegener, "On the complexity of minimizing the OBDD size for incompletely specified functions," *IEEE Transactions on TCAD*, Vol. 15, No. 11, pp. 1435-1437, 1996.

[18] C. Scholl, "Multi-output functional decomposition with exploitation of don't cares," *In Design Automation and Test Europe*, pp. 743-748, Feb. 1998.

[19] 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.

[20] 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. 7-10, 1992.