A Method to Decompose Multiple-Output Logic Functions

Tsutomu Sasao
Kyushu Institute of Technology
680-4 Kawazu
Iizuka 820-8502, Japan

Munehiro Matsuura
Kyushu Institute of Technology
680-4 Kawazu
Iizuka 820-8502, Japan

ABSTRACT

This paper shows a method to decompose a given multiple-output circuit into two circuits with intermediate outputs. We use a BDD for characteristic function (BDD for CF) to represent a multiple-output function. Many benchmark functions were realized by LUT cascades with intermediate outputs. Especially, adders and a binary to BCD converter were successfully designed. Comparison with FPGAs is also presented.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Experimentation, Theory

Keywords

Cascade, BDD, Characteristic function, FPGA

1. INTRODUCTION

Functional decomposition of logic functions [1] has wide applications, especially in the design of FPGAs [16]. Binary decision diagrams are extensively used to design such networks [5, 7, 3]. When a logic function f can be represented as $f(X_1, X_2) = g(h(X_1)X_2)$, we can design networks for $h(X_1)$ and $g(h, X_2)$ independently to implement the decomposed network shown in Fig. 1. By applying such decompositions iteratively, we can design LUT type FPGAs. Design of an LUT network for single-output logic function using functional decomposition is relatively easy. However, the design of LUT networks for multiple-output functions is not so simple. Various methods have been proposed [4, 5, 10, 12, 13, 15].

In this paper, we present a new method to decompose a multiple-output function. It uses a binary decision diagram for characteristic function (BDD for CF) [2]. This method efficiently finds the decomposition with intermediate outputs shown in Fig. 2. A recursive application of the method produces an LUT cascade with intermediate outputs as shown in Fig. 3. The LUT cascade [10] has a regular structure and is easy to design. It is a promising method to design deep sub-micron LSIs, since the interconnections are limited to the adjacent cells, and thus, the prediction of delay is easy. In a conventional FPGA, the delay of interconnections is much larger than that of logic, and this is one of the fundamental limitations on FPGA speed. On the other hand, in LUT cascades, the area for the interconnections is much smaller than conventional FPGAs.

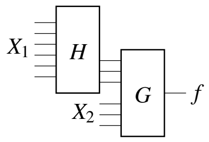


Figure 1: Conventional Functional Decomposition.

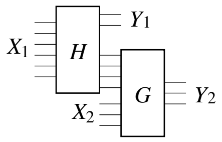


Figure 2: Functional Decomposition with Intermediate Outputs.

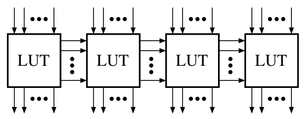


Figure 3: LUT Cascade with Intermediate Outputs.

2. DEFINITIONS AND BASIC PROPERTIES

Definition 1. *Let $F = (f_0(X), f_1(X), \ldots, f_{m-1}(X))$ be a multiple-output function. Let $X = (x_1, x_2, \ldots, x_n)$ be the input variables, and $Y = (y_0, y_1, \ldots, y_{m-1})$ be the set of variables that denotes the outputs.* **The characteristic function of a multiple-output function** *is defined as*

$$\chi(X, Y) = \bigwedge_{i=0}^{m-1} (y_i \equiv f_i(X)).$$

The characteristic function of an n-input m-output function is a two-valued logic function with $(n+m)$ inputs. It has input variables x_i $(i = 1, 2, \ldots, n)$, and output variables y_i for each output f_i. Let $B = \{0, 1\}$, $\vec{a} \in B^n$, $F(\vec{a}) = (f_0(\vec{a}), f_1(\vec{a}), \ldots, f_{m-1}(\vec{a})) \in B^m$, and $\vec{b} \in B^m$. Then, the characteristic function satisfies the relation

$$\chi(\vec{a}, \vec{b}) = \begin{cases} 1 & \text{(If } \vec{b} = F(\vec{a})\text{)} \\ 0 & \text{(Otherwise).} \end{cases}$$

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2004, June 7–11, 2004, San Diego, California, USA.
Copyright 2004 ACM 1-58113-828-8/04/0006 ...$5.00.

DEFINITION 2. *The* **support** *of a function f is the set of variables on which f actually depends.*

DEFINITION 3. *The* **BDD for CF** *for a multiple-output function $F = (f_0, f_1, \ldots, f_{m-1})$ is the ROBDD for the characteristic function χ. In this case, we assume that the root node is at the top of the BDD, and variable y_i is below the support of f_i, where y_i is the variable representing f_i.*

DEFINITION 4. *In a BDD for CF, for each node that represents an output y_i, remove the node and the edge pointing the constant 0 node, and redirect the edge to the other child of y_i. Apply this operation to all the nodes that represent the output y_i. This operation is denoted by* **removal of the output variables y_i by shorting**.

Let the height of the root node be the total number of variables, and let the height of the constant nodes be 0.

DEFINITION 5. *The* **width of a BDD at height k**, *is the number of edges crossing the section of the BDD between variables z_k and z_{k+1}, where edges incident to the same node are counted as one.*

The next theorem is the key result of the paper. It is similar to that of [5], but finds a decomposition with intermediate outputs as shown in Fig. 2.

THEOREM 1. *Let (X_1, Y_1, X_2, Y_2) be the variable ordering of the BDD for CF, 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. 2. Then, the necessary and sufficient number of connections between two blocks H and G is $\lceil \log_2 W \rceil$.*

PROOF. By the definition of the BDD for CF, it is clear that we can realize functions for Y_1 and Y_2 by the network shown in Fig. 2 . In the BDD for CF, remove the nodes that represent the outputs Y_1 by shorting, and we have the BDD for CF that represents the multiple-output functions Y_2. Note that this operation does not change the width of the BDD. Let W be the width of the BDD for CF at the height $(n_2 + m_2)$ after the removal of the output variables Y_1 by shorting. Consider the decomposition chart for the decomposition $g(h(X_1), X_2)$. The column multiplicity is equal to W. In other words, if we ignore the outputs in Y_1, $\lceil \log_2 W \rceil$ lines are necessary and sufficient to realize the functions in Y_2. Since Y_1 depends only on X_1 and does not influence on the number of connections between H and G, the necessary and sufficient number of wires between networks H and G is $\lceil \log_2 W \rceil$. □

Let (X_1, Y_1, X_2, Y_2) be the variable ordering of a BDD for CF, where $Y_1 = (y_0, y_1, \ldots, y_{k-1})$. Realize functions $f_i(X_1)$ $(i = 0, 1, \ldots, k-1)$ by the network H in Fig. 2. Let W be the width of the BDD for CF at the height $n_2 + m_2$. To W nodes, assign different binary numbers of $u = \lceil \log_2 W \rceil$ bits. Let $h_1, h_2, \ldots, h_u$ be the functions realized by the lines that connect two blocks. Then, the output functions $(f_k, f_{k+1}, \ldots, f_{m-1})$ can be represented as functions of $(h_1, h_2, \ldots, h_u, X_2)$. Also, the BDD for CF can be represented as shown in Fig. 4.

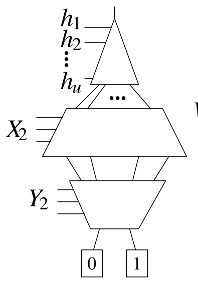


Figure 4: Realization of Y_2 by BDD for CF.

EXAMPLE 1. *Let us design the two-bit adder (ADR2). The relations of inputs and outputs of ADR2 are:*

$$
\begin{array}{rccc}
 & & a_1 & a_0 \\
+) & & b_1 & b_0 \\
\hline
 & s_2 & s_1 & s_0
\end{array}
$$

Thus, we have,

$$
\begin{aligned}
s_0 &= a_0 \oplus b_0 \\
s_1 &= a_0 b_0 \oplus (a_1 \oplus b_1) \\
s_2 &= a_0 b_0 (a_1 \vee b_1) \vee a_1 b_1.
\end{aligned}
$$

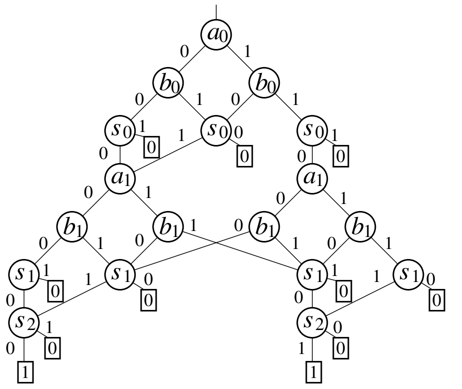
Consider the partition of the variables: $X_1 = (a_0, b_0)$, $Y_1 = (s_0)$, $X_2 = (a_1, b_1)$, and $Y_2 = (s_1, s_2)$. In this case, we use the variable ordering $(X_1, Y_1, X_2, Y_2) = (a_0, b_0, s_0, a_1, b_1, s_1, s_2)$. Fig. 5(a) shows the BDD for CF. Let the partition the variables be $X = (X_A, X_B)$, where $X_A = (X_1, Y_1)$, and $X_B = (X_2, Y_2)$. Then, the width W of the BDD at the height four is two. Thus, only one line is necessary to connect two blocks H and G, since $\lceil \log_2 W \rceil = 1$.

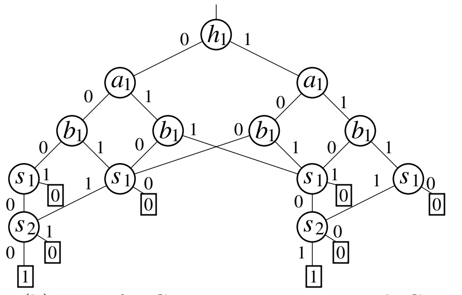
Note that s_0 is a function of variables in X_1. Next, introduce an intermediate variable h_1, and replace the top part of the BDD with the decision tree that has h_1 as the control variable (Fig. 5(b)). Then, as shown in Fig. 5(c), construct the MTBDD that has h_1, a_1, and b_1 as inputs, and s_1, and s_2 as outputs. Finally, we can obtain the network for adr2 as shown in Fig. 6. (*End of Example*)

3. OUTLINE OF THE DESIGN ALGORITHM FOR LUT CASCADES

In this section, we briefly describe a method to design an LUT cascade by using a BDD for CF.

By iterative application of functional decompositions, we can generate LUT cascades. Let the number of inputs of an LUT be $k \geq 3$. Given a multiple-output function, generate the BDD for CF, and minimize it. Then, select k input variables that are near to the root node. Next, obtain W, the width of the BDD. In this case, ignore the edges that connect nodes for the outputs and the constant 0. Then, introduce $u = \lceil \log_2 W \rceil$ intermediate variables. Next, assign binary codes of u bits to the W sub-functions. In this case, we use the simplest strategy: For each sub-function, assign one binary code; do not consider *don't care*; and assign unused codes to a certain sub-function. By using an LUT, realize k-input $(u + w)$-output function, where w denotes the number of output variables in the selected variables. Replace k variables by u intermediate variables, and re-construct the BDD for CF. Then, again, select k-variables that are near


(a) BDD for CF representing adder.


(b) BDD for CF representing network G.

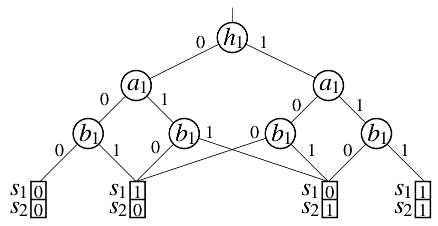

(c) MTBDD representing network G.

Figure 5: Design of two-bit adder.

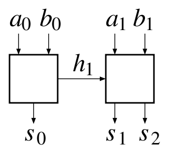


Figure 6: Two-bit adder (ADR2).

the root node, and do similar operations until all the variables are selected.

4. DETAILED DESIGN ALGORITHM FOR LUT CASCADES

For practical multiple-output functions, BDDs for CFs are often too large to represent all the outputs at one time. Also, even if the BDD for CF is stored in a memory of a computer, it can be too large to be realized by an LUT cascade. In such a case, we partition the outputs into groups, and for each group of outputs, we design an LUT cascade to realize the functions in the group.

We partition the outputs so that each set of outputs depends on as small number of input variables as possible. This will reduce the size of the BDD for CF. Thus, at first, by using Algorithm 1, we reorder the output functions so that the support will increase as slowly as possible. Second, we find an ordering of the input and the output variables by Algorithm 2 to construct a BDD for CF. Third, we generate cascade from the BDD for CF by Algorithm 3. And, finally, we partition the outputs into groups by Algorithm 4. For each group, we increase the number of outputs one by one while the functions are realizable with an LUT cascade. All the algorithms in this section are heuristic ones.

4.1 Ordering of Outputs

ALGORITHM 1. *(Ordering of the Output Functions)* *Let* $(f_0, f_1, \ldots, f_{m-1})$ *be the initial order of the output functions.*

```
1. $i \leftarrow 0$, $j \leftarrow 0$, $minT \leftarrow \infty$, $minorder \leftarrow$ Initial order.
2. Exchange the positions for $f_i$ and $f_j$.
3. Compute the value of $T \leftarrow \sum_{k=0}^{m-1} |\bigcup_{l=0}^{k} sup(f_l)|$, where $sup(f_l)$ denotes the support of $f_l$.
4. If $T < minT$, then $minT \leftarrow T$, and $minorder \leftarrow$ Current output order.
5. If $j < m-1$, then $j \leftarrow j+1$, and go to Step 2.
6. If $j \leftarrow 0$ and $i < m-1$, then $i \leftarrow i+1$, and go to Step 2.
7. If $minT$ is updated, then $i \leftarrow 0$, and go to Step 2. Else let $minorder$ be the output order, and terminate.
```

In Step 3, T denotes the number of variables in the support of the group of functions. Algorithm 1 tries to find the ordering of the outputs that increases the sizes of supports as slowly as possible.

4.2 Ordering of Variables

By using the BDD for CF, we decompose the function in the form $g(h_1(Z_1), h_2(Z_1), \ldots, h_u(Z_1), Z_2)$, where Z_1 and Z_2 denote sets of input and output variables.

If $\{Z_1\}$ includes any output variables, then the block H in Fig. 2 produces external outputs that correspond to the output variables. This will reduce the number of inputs to the block G in Fig. 2. Therefore, as an initial variable ordering of the BDD for CF, we try to find the ordering so that many output variables are near to the root nodes, while keeping the width of the BDD smaller than a certain value.

ALGORITHM 2. *(Ordering of the Variables)*

```
1. By Algorithm 1, obtain the output order $(f_0, f_1, \ldots, f_{m-1})$.
2. Let $sup(f_i)$ be the support of the function $f_i$, and $y_i$ be the output variables for $f_i$.
3. From the root node of the BDD, let the ordering of the variables be $sup(f_0), y_0, sup(f_1)-sup(f_0), y_1, sup(f_2)-sup(f_1)-sup(f_0), y_2, \ldots, sup(f_{m-1})-sup(f_{m-2})-\cdots-sup(f_0), y_{m-1}$. In this case, the variable ordering within $sup(f_i)$ is the same as one in the minimum SBDD.
```

We use this ordering as an initial variable ordering of the BDD for CF, and optimize the variable order by sifting algorithm [9], where the sum of widths is used as the cost function of the BDD.

4.3 Derivation an LUT Cascade from a BDD for CF

In this part, we show an algorithm to derive an LUT from a BDD for CF. Let k be the maximum number of inputs for an LUT, and r be the maximum number of outputs of an LUT. Let (Z_1, Z_2) be a partition of variables, and let the given function be decomposed as $f(Z) = g(h_1(Z_1), \ldots, h_u(Z_1), Z_2)$. Let $Z_1 = (X_1, Y_1)$ and $Z_2 = (X_2, Y_2)$, where X_1 and X_2 denote the sets of input variables, and Y_1 and Y_2 denote the sets of output variables. Let W be the width of the BDD at the height $|Z_2|$. If $|X_1| \leq k$, and if $(Y_1 + \lceil \log_2 W \rceil) \leq r$, then the function can be realized by an LUT cascade, where $\{Z_1\}$ is a bound set.

Algorithm 3. *(Derivation of an LUT Cascade from a BDD for CF)*
This algorithm finds a partition of the set of variables for CF, when the variable order and widths of BDD for CF are given. Let $F = (f_0, f_1, \ldots, f_{m-1})$ be a given multiple-output function; Z be the support of CF; k be a maximum number of inputs for an LUT; r be a maximum number of outputs for an LUT; z_i be the variable whose height is i; and $\{Z_t\}$, $\{Z_a\}$, $\{Z_b\}$, $\{Z_c\}$, $\{Z_{in}\}$, and $\{Z_{out}\}$ be sets of variables.

```
1. $i \leftarrow |Z|$, $Z_t \leftarrow Z$, $\{Z_{in}\} \leftarrow \phi$, $\{Z_c\} \leftarrow \phi$, $l \leftarrow 1$.
2. While $\{Z_t\} \neq \phi$, do Steps (a)–(d).
   (a) $j \leftarrow i - 1$, $top \leftarrow j$, $\{Z_{out}\} \leftarrow \phi$.
   (b) While $i > 0$ and $|Z_{in}| \leq k$, do Steps i–iv.
       i. If $z_i$ is an output variable for CF then $\{Z_{out}\} \leftarrow \{Z_{out}\} \cup \{z_i\}$, else $\{Z_{in}\} \leftarrow \{Z_{in}\} \cup \{z_i\}$.
       ii. $u_i \leftarrow \lceil \log_2 w_i \rceil$, where $w_i$ is width of BDD for CF at the height $i$.
       iii. If $u_i < k$ and $|Z_{out}| + u_i \leq r$ then $j \leftarrow i$ and $\{Z_a\} \leftarrow \{Z_{in}\} \cup \{Z_{out}\}$.
       iv. $i \leftarrow i - 1$.
   (c) If $j = top$ then the function cannot be realized by an LUT cascade, and terminate.
   (d) $\{Z_b\} \leftarrow \{Z_t\} - \{Z_a\}$, $\{Z_l\} \leftarrow \{Z_a\} - \{Z_c\}$. Let $\{H\}$ be a set of intermediate variables for decomposition $g(h(Z_a), Z_b)$. $\{Z_t\} \leftarrow \{Z_b\}$, $\{Z_{in}\} \leftarrow \{H\}$, $i \leftarrow j - 1$, $\{Z_c\} \leftarrow \{H\}$, $l \leftarrow l + 1$.
3. For the partition $(Z_1, Z_2, \ldots, Z_{l-1})$, realize an LUT cascade.
```

Let $\{Z_1\}$ be the bound set, and μ be the column multiplicity of the decomposition. Then, the decomposition of a BDD for CF produces $u = \lceil \log_2 \mu \rceil$ intermediate variables, and possibly some external outputs variables that are contained in $\{Z_1\}$.

4.4 Derivation of an LUT Cascade for a Multiple-Output Function

Here, we will give an algorithm to derive an LUT cascade for a given multiple-output function. Note that this algorithm partitions the outputs into groups, then generate BDD for CF for each group, and realize each group by an LUT cascade.

Algorithm 4. *(Derivation of a set of LUT Cascades for a Multiple-Output Function)*

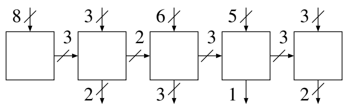
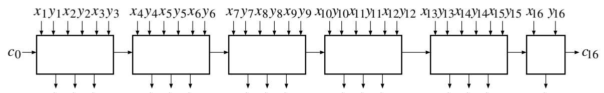


Figure 7: LUT Cascade for vg2.



$x_1 y_1 x_2 y_2 x_3 y_3$
$x_4 y_4 x_5 y_5 x_6 y_6$
$x_7 y_7 x_8 y_8 x_9 y_9$
$x_{10} y_{10} x_{11} y_{11} x_{12} y_{12}$
$x_{13} y_{13} x_{14} y_{14} x_{15} y_{15}$
x_{16} y_{16}
c_0
c_{16}
s_1 s_2 s_3
s_4 s_5 s_6
s_7 s_8 s_9
s_{10} s_{11} s_{12}
s_{13} s_{14} s_{15}
s_{16}

Figure 8: LUT Cascade for 16-bit adder for $k = 7$.

```
1. By Algorithm 1, obtain the order of the output functions, and let it be $(f_0, f_1, \ldots f_{m-1})$. Let $F_a$ be a set of functions.
2. $i \leftarrow 0$, $F_a \leftarrow \phi$.
3. Construct a BDD for CF that represents $F_a \cup \{f_i\}$, and optimize the variable ordering. Use Algorithm 2 to find an initial ordering of the variables.
4. By using Algorithm 3, try to realize an LUT cascade.
5. When the cascade is realizable. If $i = m$, then generate the LUT for $F_a$ and terminate, else $F_a \leftarrow F_a \cup \{f_i\}$, $i \leftarrow i + 1$, and go to Step 3.
6. When the cascade is not realizable. If $|F_a| = 0$, then the function cannot be realized by LUT cascades, and terminate. Else, produce the LUT cascade for $F_a$. $F_a \leftarrow \phi$, and go to Step 3.
```

5. EXPERIMENTAL RESULTS

5.1 LUT Cascade

We implemented Algorithm 4 in the C programming language, and designed LUT cascades for selected MCNC89 benchmark functions. Table 1 shows the experimental results. In the table, *Name* denotes the name of benchmark function; *In* denotes the number of inputs; *Out* denotes the number of outputs; *Size of BDD for CF* denotes the number of nodes to represent the multiple-output function. *LUT* denotes the total number of outputs used in the LUTs; *Lvl* denotes the maximum number of levels; *Cas* denotes the number of cascades; *Time* denotes the time (sec) to generate LUT cascades from SBDDs; and k denotes the maximum number of inputs of the LUTs. In this experiment, r is set to a sufficiently large value. Also, the symbol $-$ denotes that Algorithm 4 failed to produce a cascade. In Table 1, we only showed the functions where we could construct monolithic BDDs for CFs. We used an IBM PC/AT compatible machine using a Pentium4 3.2GHz processor with 1GByte of memory. The operating system was Windows XP, and we used gcc complier on *cygwin*.

Fig. 7 shows the LUT cascade for the benchmark function *vg2*, where $k = 8$. It uses five cells and 21 LUT outputs. 19 functions with 8 inputs, and two functions with 6-input.

Figs. 8 and 9 show the LUT cascades for *my_adder*, for $k = 7$ and $k = 9$, respectively. Note that *my_adder* is a 16-bit adder with a carry input, and the algorithm successfully found optimal ripple-carry adders.

Table 1: Cascade Realizations of MCNC89 Benchmark Functions.

Name	In	Out	Size of BDD for CF	$k=8$				$k=9$				$k=10$				FPGA	
				LUT	Lvl	Cas	Time	LUT	Lvl	Cas	Time	LUT	Lvl	Cas	Time	LUT	Delay
C432	36	7	1972	141	17	2	–	113	16	1	2.0	76	11	1	2.0	172	29.4
C880	60	26	800877	–	–	–	–	–	–	–	–	373	19	4	555.6	155	25.0
alu2	10	6	258	11	2	1	0.0	10	2	1	0.0	6	1	1	0.0	115	17.2
alu4	14	8	1532	56	6	2	0.1	46	5	2	0.2	24	3	1	0.3	386	19.7
apex1	45	45	4897	270	23	2	30.1	167	19	1	291.1	115	12	1	291.7	936	25.3
apex2	39	3	449	54	12	1	0.4	35	8	1	0.4	35	8	1	0.4	167	18.6
apex3	54	50	3401	250	21	2	138.9	164	19	1	318.0	129	14	1	317.9	773	26.0
apex4	9	19	2339	40	2	3	0.3	19	1	1	8.5	19	1	1	8.4	1276	37.3
apex7	49	37	5064	198	24	2	25.7	156	19	1	52.0	110	13	1	52.1	110	16.5
b9	41	21	804	51	9	1	3.4	42	7	1	3.4	38	6	1	3.4	50	12.6
cc	21	20	323	30	4	1	0.7	30	4	1	0.7	27	3	1	0.6	23	13.0
cht	47	36	700	63	10	1	5.8	56	8	1	5.9	53	7	1	5.7	39	13.3
cm150a	21	1	52	11	4	1	0.0	9	4	1	0.0	8	3	1	0.0	9	11.7
comp	32	3	58	11	5	1	0.0	11	5	1	0.0	9	4	1	0.0	52	16.3
count	35	16	149	26	6	1	0.2	26	6	1	0.2	24	5	1	0.2	54	15.4
duke2	22	29	822	59	7	1	4.6	49	5	1	4.6	45	4	1	4.6	248	19.3
e64	65	65	260	74	10	1	6.5	72	8	1	6.3	72	8	1	6.3	268	16.9
example2	85	66	6347	308	41	1	202.0	215	27	1	932.9	166	19	1	933.5	161	15.5
frg1	28	3	132	18	6	1	0.0	15	5	1	0.0	13	4	1	0.0	44	16.7
lal	26	19	310	30	5	1	0.3	28	4	1	0.3	26	4	1	0.3	31	11.3
misex2	25	18	280	29	5	1	0.3	26	4	1	0.2	25	4	1	0.2	52	13.1
mux	21	1	52	11	4	1	0.0	9	4	1	0.0	8	3	1	0.0	9	11.7
my_adder	33	17	149	22	6	1	0.1	20	4	1	0.1	20	4	1	0.1	57	19.5
pcler8	27	17	992	52	8	2	0.6	57	8	1	2.6	43	6	1	2.6	41	16.5
seq	41	35	1554	152	20	1	28.6	102	13	1	28.5	83	9	1	28.4	1031	25.2
term1	34	10	832	118	18	1	2.3	65	10	1	2.4	49	8	1	2.3	51	15.3
too_large	38	3	449	54	12	1	0.5	35	8	1	0.4	35	8	1	0.4	1815	36.5
ttt2	24	21	3529	72	9	2	1.5	73	9	2	3.2	54	6	1	3.2	69	14.3
unreg	36	16	225	34	7	1	0.2	31	6	1	0.2	28	5	1	0.1	34	14.1
vg2	25	8	155	19	5	1	0.1	18	4	1	0.0	16	4	1	0.0	40	14.5

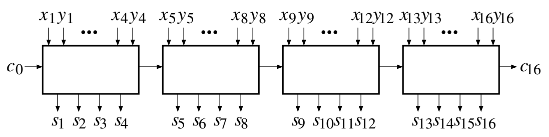


Figure 9: LUT Cascade for 16-bit adder for $k = 9$.

5.2 Comparison with FPGAs

To compare our approach with FPGAs, we used *Synplify* from Synplicity, Inc., Sunnyvale, CA. for logic synthesis, and *ISE Foundation* for mapping into Xilinx Virtex (0.22μm, 5-layer metal, 2.5V) XCV50-6 (180pin) FPGAs. In Table 1, the columns headed by FPGA denote the design results of FPGAs. *LUT* denotes the number of 4-input LUTs, and *Delay* denotes the estimated delay (ns). Note that the number of LUTs does not show the real chip area. In FPGAs, more than 90% of the chip area is devoted to interconnections [8]. For functions with many outputs, LUT cascades are slower than FPGAs. So, for such circuits, the outputs must be partitioned into smaller groups.

5.3 Other Functions

5.3.1 RGB Color Converter

This circuit computes $U = -0.619R - 0.3316G + 0.5B$, where R, G and B are represented by 8 bits, and U is represented by 9 bits, and the most significant bit is the sign bit. An LUT cascade with $k = 13$ is shown in Fig. 10. The FPGA design required 12818 4-input LUTs and 78.7 ns of delay for mapping into Xilinx Virtex XCV600-6 (316pin). In this case, we used *Synplify* without speed priority options;

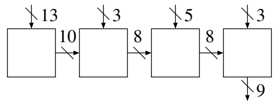


Figure 10: LUT Cascade for RGB Color Converter.

when we used the speed options, *Synplify* did not finish in 22 hours. For this kind of application, LUT cascades are much faster than standard FPGAs, since the delay time of a cell of the LUT cascade is at most 4 ns.

5.3.2 Binary to BCD Converter

This circuit converts a 16-bit binary number into a 5-digit BCD number. Among various implementations, Muroga [7] shows a circuit using 13 modules (ROMs). Algorithm 4 generated the cascade in Fig. 11, which uses only three cells of 11 inputs each. The input binary number is represented by $x_1, x_2, x_3, \ldots, x_{15}, x_{16}$ and the output BCD number is represented by $f_1, f_2, f_3; f_4, f_5, f_6, f_7; f_8, f_9, f_{10}, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16}, f_{17}, f_{18}, f_{19}$. Note that f_0, the most significant bit of the most significant digit, is always 0, so, it is omitted. Also, $f_{19} = x_{16}$, that is, the least significant output is equal to the least significant input.

When the specification of the converter was given by an algorithm written in Verilog, we had a FPGA with 695 4-input LUTs and 70.7 ns delay for Xilinx Virtex XCV150-6 (260pin). On the other hand, when the specification of the circuit was given by a BDD, and then each node of the BDD is replaced by a multiplexer, we had a FPGA with 1659 4-input LUTs and a 33.1 ns delay. Also for this application, the LUT cascade is faster than FPGA realizations.

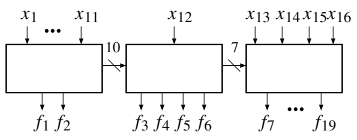


Figure 11: LUT Cascade for 16-bit Binary to BCD Converter.

6. LIMITATION OF THE APPROACH

6.1 Limitation due to the Data Structure

The most time-consuming step of the algorithm is the optimization of BDDs for CFs. The size and optimization time of BDDs for CFs are, in most cases, larger than those of SBDDs. When the size of the BDD for CF is too large to build, we have to partition the functions into smaller groups so that each BDD for CF can be constructed.

Logic functions having compact BDD representations include, symmetric functions, adders, and comparators. On the other hand, randomly generated functions and multiplier have BDDs with exponential size (in n, the number of input variables.), and they cannot be designed by our method when n is large.

6.2 Limitation due to the Network Structure

In Table 1, experimental results for $k = 8$ to 10 are shown. When $k = 5$, most functions in Table 1 cannot be realized by LUT cascades. This is due to the fact that a given function f is realized by a cascade of k-input LUTs only if $\lceil \log_2 W \rceil \leq k - 1$, where W is the width of the BDD for f in the decomposition level.

In the design method for the cascade, each input variable can appear in the input terminal of the cascade only once. If we remove this restriction, then an arbitrary m-output logic function can be realized by an LUT cascade of $(m+2)$-input cells [11].

Also, the algorithm tries to realize a given multiple-output function by using as few cascades as possible. However, this strategy may not be practical in some applications. When the number of the outputs is large, we should partition the outputs into smaller groups and realize each group by an independent cascade. This strategy often produces cascades with fewer LUTs, but the wiring will be more complex.

7. CONCLUDING REMARKS

In this paper, we presented a method to decompose a multiple-output logic function by using a BDD for CF. This method efficiently produces LUT cascades with intermediate outputs.

The decomposition method presented in this paper is quite fundamental, and is promising not only for LUT cascades, but also for random LUT networks. Extension to incompletely specified functions is a challenging problem.

Acknowledgments

This research is partly supported by JSPS, the Grant in Aid for Scientific Research, and MEXT, the Kitakyushu area innovative cluster project.

8. REFERENCES

[1] R. L. Ashenhurst, “The decomposition of switching functions,” *In Proceedings of an International Symposium on the Theory of Switching*, pp. 74-116, April 1957.

[2] P. Ashar and S. Malik, “Fast functional simulation using branching programs,” *Proc. International Conference on Computer Aided Design*, pp. 408-412, Nov. 1995.

[3] Ting-Ting Hwang, R. M. Owens, M. J. Irwin, and Kuo Hua Wang, “Logic synthesis for field-programmable gate arrays,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, Vol. 13, No. 10, pp. 1280-1287, Oct. 1994.

[4] J.-H. R. Jian, J.-Y. Jou, and J.-D. Huang, “Compatible class encoding in hyper-function decomposition for FPGA synthesis,” *Design Automation Conference*, pp. 712-717, June 1998.

[5] Y-T. Lai, M. Pedram and S. B. K. Vrudhula, “BDD based decomposition of logic functions with application to FPGA synthesis,” *30th ACM/IEEE Design Automation Conference*, June 1993.

[6] S. Muroga, *VLSI System Design*, John Wiley & Sons, 1982, pages 293-306.

[7] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Logic Synthesis for Field-Programmable Gate Arrays*, Kluwer, 1995.

[8] J. Rose, R. J. Francis, D. Lewis, and P. Chow, “Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency,” *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 5, pp. 1217-1225, Oct. 1990.

[9] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” *Proc. ICCAD-93*, pp. 42–47, 1993.

[10] T. Sasao, M. Matsuura, and Y. Iguchi, “A cascade realization of multiple-output function for reconfigurable hardware,” *International Workshop on Logic and Synthesis,* Lake Tahoe, CA, June 12-15, 2001, pp.225-230.

[11] T. Sasao, “Design methods for multi-rail cascades,” *International Workshop on Boolean Problems*, Freiberg, Germany, Sept. 19-20, 2002, pp. 123-132.

[12] H. Sawada, T. Suyama, and A. Nagoya, “Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization,” *Proc. ICCAD*, pp. 353-359, Nov. 1995.

[13] C. Scholl and P. Molitor, “Communication based FPGA synthesis for multi-output Boolean functions,” *Asia and South Pacific Design Automation Conference*, pp. 279-287, Aug. 1995.

[14] http://www.synplicity.com

[15] B. Wurth, K. Eckl, and K. Anterich, “Functional multiple-output decomposition: Theory and implicit algorithm,” *Design Automation Conf.*, pp. 54-59, June 1995.

[16] http://www.xilinx.com