社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE

レイル数に制限のある LUT カスケードの論理合成法 ―多出力論理関数の直接実現―

アラン　ミシュチェンコ[†]　　笹尾　勤[‡]

†ポートランド州立大学電気計算機工学科 米国コロラド州
‡九州工業大学　情報工学部，マイクロ化総合技術センター 〒820-8502 飯塚市大字川津 680-4
E-mail:　†alanmi@ece.pdx.edu,　‡sasao@cse.kyutech.ac.jp

あらまし　多出力論理関数を実現する LUT カスケード回路の実現法について述べる．多出力関数の表現法として，幾つかの方法について述べ，それを OR 分解する方法について示す．次に，レイル数（二つのセルの間の線数）に制限がある場合のカスケード回路の設計アルゴリズムを示す．これより，メモリ量の削減が可能であり，大規模かつ高速な回路の実現も可能となる．多数のベンチマーク関数に本手法を適用した結果を述べる．
.

キーワード　プログラマブルロジック，ルックアップテーブル（LUT），論理合成，関数分解，二分決定グラフ

Logic Synthesis of LUT Cascades with Limited Rails ― A Direct Implementation of Multi-Output Functions ―

Alan MISHCHENKO[†]　and　Tsutomu SASAO[‡]

† Department of ECE, Portland State University, P.O. Box 751, Portland 97207, Oregon, USA
‡ Center for Microelectronic Systems and Department of CSE, Kyushu Institute of Technology, Iizuka, Fukuoka, 820-8502 JAPAN
E-mail:　† alanmi@ece.pdx.edu,　‡ sasao@cse.kyutech.ac.jp

Abstract　Programmable LUT cascades are used to evaluate multi-output Boolean functions. This paper shows several representations of multi-output functions and introduces a new decomposition algorithm applicable to these representations. The algorithm produces LUT cascades with the limited number of rails, which leads to significantly faster circuits and applicability to large designs. The experiment shows that the proposed algorithm performs well on benchmark functions.

Keyword　Programmable Logic, Look-Up Table (LUT), Logic Synthesis, Decomposition, Binary Decision Diagrams.

1. Introduction

In recent years, programmable logic devices receive more attention due to their improved performance, flexibility, and low production cost. An important aspect influencing the performance of these devices is the speed of evaluation of complex logic functions, which are programmed in them.

Several approaches to the fast evaluation of logic functions are known, in particular, implementing them in hardware (FPGAs, CPLDs) and realizing them in software (branching programs [1]).

In this paper, we discuss the third option, proposed in [16] and further developed in [7][10][14]. This approach evaluates a logic function using a series of fast memory lookups. To this end, the Boolean function is implemented as a cascade of lookup tables (LUTs). To find the value of the function for an assignment of the input variables, the LUTs in the cascade are evaluated in a sequence. The address word applied to each LUT is composed of the values of the external input variables and the output values of the previous LUTs. The last LUT in the cascade produces the value of the function. A comprehensive survey of different types of cascades and their expressive power can be found in [14].

Logic synthesis for the LUT cascade consists in determining the actual contents of LUTs for the cascade to realize the given multi-output logic function. Efficient synthesis methods have been developed recently for this purpose. In particular, [15] discusses functional representations for LUT cascade synthesis; [16] discusses synthesis with no limit on rails; [7] and [10] discuss memory encoding to reduce the number of LUTs.

The above synthesis methods work well when arbitrary large LUTs are available. However, the functions to be implemented may be beyond the capacity of available LUTs, or the size of LUTs may be limited for practical reasons. In this case, some kind of decomposition is needed to fit large functions into narrow cascades.

In this paper, we consider the LUT cascade synthesis with the limit on the number of rails. The present work makes the synthesis methodology more versatile and robust. With a reasonable overhead in the number of LUTs, it becomes applicable to designs of any size.

The rest of the paper is organized as follows. Section 2 presents related terminology. Section 3 briefly reviews the LUT cascade logic synthesis flow. Section 4 discusses several representations of multi-output Boolean functions and relationship among them. Section 5 presents the main contribution of the paper, a decomposition method to produce limited-rail cascades. Section 6 shows experimental results, and Section 7 concludes the paper.

2. Terminology

The functions considered in this paper are completely-specified Boolean functions, unless stated otherwise. The reader is assumed to be familiar with the basic concepts of Binary Decision Diagrams (BDDs) [3].

A k-input *Look-Up Table* (LUT) implements any single-output function of k input variables. A k-input u-output *memory cell* composed of u k-input LUTs evaluated in parallel implements any k-input u-output function. A LUT cascade is a sequence of s memory cells evaluated one by one.

The wires connecting the outputs of each memory cell with the inputs of the next cell are called *rails*. Memory cells with u outputs produce a u-rail LUT cascade. Additional k-u inputs (*side variables*) of each LUT are set to the values of the external input variables. The first cell has all its k inputs set to the values of the external input variables. The last cell produces the outputs of the multi-output function. A u-rail LUT cascade of sufficient length can simultaneously evaluate up to u outputs of the multi-output function, as shown in Fig. 1.

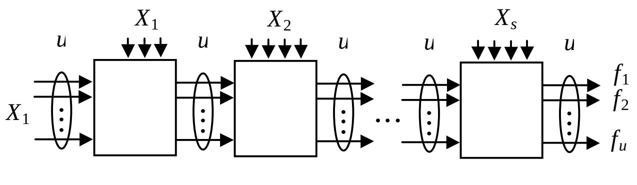

Figure 1. A u-rail LUT cascade with s memory cells.

The sets of variables (X_1, X_2, … X_s), whose values are fed into each cell can be either disjoint or non-disjoint giving rise to two types of cascades, irredundant or redundant [14]. In this paper, we consider irredundant cascades. The partitioning of variables into disjoint sets (X_1, X_2, … X_s) is determined by the variable order in the decision diagram used to synthesize the cascade.

3. LUT Cascade Synthesis Flow

The LUT cascade design process can be divided into several steps, shown in Fig. 2. In this paper, we deal with the OR-decomposition step, which was not considered in previous publications on LUT cascade synthesis.

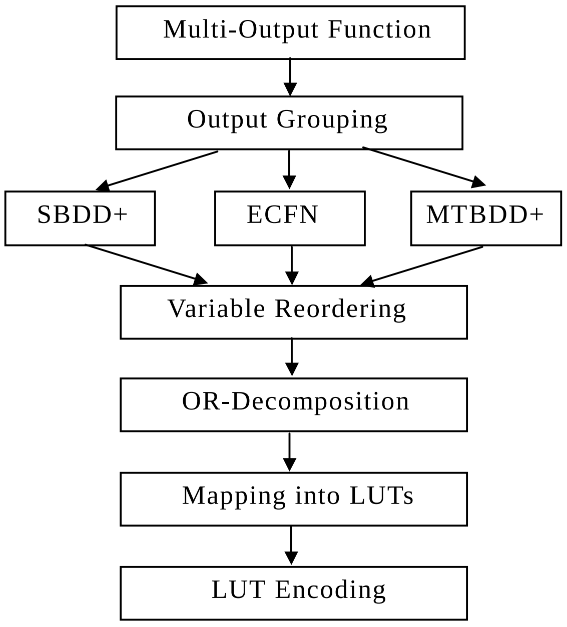

Figure 2. The outline of LUT cascade synthesis flow.

4. Representations of Multi-Output Functions

This section presents an overview of representations of multi-output functions used in LUT cascade synthesis. A detailed discussion and experimental results comparing these representations can be found in [15].

4.1. SBDD+

A single-output function is represented by the BDD [3]. A multi-output function is a set of single-output functions and can be represented by the Shared BDD (SBDD). The SBDD is a more compact compared to the set of unrelated BDDs because it shares the isomorphic subfunctions belonging to different outputs.

Definition 1. [15] SBDD+ of a u-output function is a single-output function constructed as follows:

1) Introduce $d = \lceil \log_2 u \rceil$ auxiliary variables above the variables used in the SBDD.
2) Encode each output i of the function by a unique minterm m_i, depending on the auxiliary variables.
3) Add all the products of the output functions by the corresponding encoding minterms.

The number of nodes in the SBDD+ is larger than that in the SBDD because of the additional (possibly incomplete) binary tree, which selects one output of the multi-output function.

4.2. MTBDD+

A Multi-Terminal Binary Decision Diagrams (MTBDD) [4] can represent a binary-input integer-valued-output function. MTBDDs enjoy the same remarkable properties as the BDDs, for example, canonicity.

A multi-output function can be represented by an MTBDD. For this purpose, we create a binary-input integer-valued-output function, with the number of output values equal to the number of different combinations of the output values in the multi-output function, under all possible combinations of the input variables.

The MTBDD can evaluate all outputs of the multi-output function at the same time; however the size of MTBDD is in many cases larger than that of SBDD or SBDD+.

Definition 2. The MTBDD+ of a *u*-output function is a single-output function constructed as follows:

1) Create the MTBDD for the *u*-output function. This MTBDD has at most 2^u terminal nodes.
2) Introduce $d = \lceil \log_2 u \rceil$ auxiliary variables below the variables used in the MTBDD.
3) For each terminal node of the MTBDD, create a unique *u*-variable function depending on the auxiliary variables. Such unique functions can always be created because there are $N = 2^{2^d} \geq 2^u$ different Boolean functions depending on *d* auxiliary variables.
4) Compute the sums of paths in the MTBDD leading to each terminal node.
5) Find the sum of products of each set of paths by the corresponding unique function.

It is possible to derive an MTBDD+ that is functionally equivalent to an SBDD+, but unlike the SBDD+, the MTBDD+ has auxiliary variables ordered at the bottom. An MTBDD+ and an SBDD+ are functionally equivalent if the encoding of the outputs in the SBDD+ is compatible with the assignment of unique functions to the terminal nodes in the MTBDD+.

Here is one way to make these encodings compatible:

1) In the SBDD+, select the codes for the output functions to be equal to the binary representation of the integer numbers of each outputs.
2) In the MTBDD+, select the unique functions for each terminal node in such a way that the value of these functions in each minterm is equal to the value of the corresponding output in the given terminal node.

4.3. ECFN

Encoded Characteristic Function of Non-zero outputs (ECFN) [13] is another representation of multi-output functions using auxiliary variables. ECFN is derived in the same way as SBDD+ or MTBDD+, without restriction on the order of the auxiliary variables. The relationship among these three representations is shown in Fig 3.

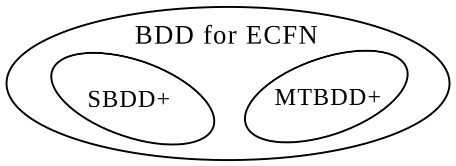

Figure 3. Relationship of BDD for ECFN, SBDD+, and MTBDD+.

Because of the freedom to order the auxiliary variables, the BDD for the ECFN has typically fewer nodes and smaller width compared to both SBDD+ and MTBDD+ [15][16]. The disadvantage of the ECFN is that each output of the given function is evaluated independently.

Therefore, in the following sections, we concentrate on MTBDDs. We address the issue of their potentially large size by developing specialized decomposition methods.

Example 1. Consider a 1-bit adder with inputs *a* and *b*, and outputs s_0 and s_1, $s_0 = a \oplus b$, $s_1 = ab$. The ECFN of the adder is: $F = \bar{z}\,(a \oplus b) + zab$, where *z* is the auxiliary variable. The SBDD and the MTBDD of the adder are shown in Fig. 4. The SBDD+, the BDD for ECFN, and the MTBDD+ are shown in Fig. 5.

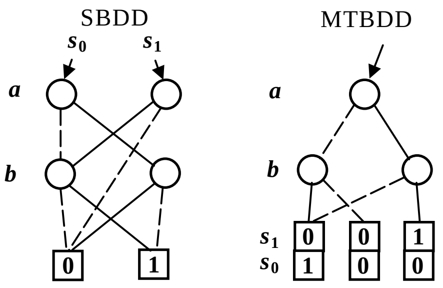

Figure 4. SBDD and MTBDD for 1-bit adder.

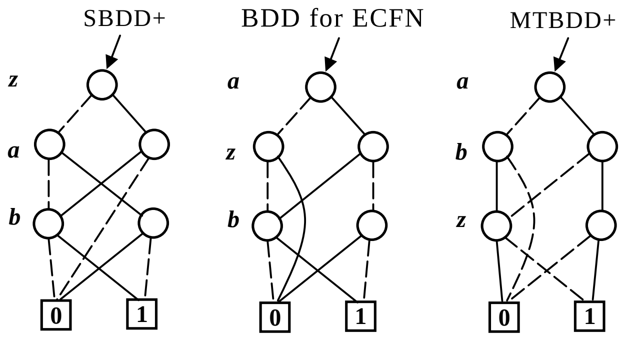

Figure 5. SBDD+, BDD for ECFN, and MTBDD+ for 1-bit adder.

5. OR-Decomposition of Decision Diagrams

5.1. Background

Definition 1. The *support* of the function f is the set of variables X, which influence the output value of the function.

In this paper, we consider a fixed order of the support variables determined by its order in the decision diagram representing f.

Definition 2. For the function f and a subset of its support variables, X_1, the set of all different *cofactors*, $\{q_1(X), q_2(X),.. q_\mu(X)\}$, of f with respect to (w.r.t.) X_1 is derived by substituting all possible assignments of variables X_1 into f and deleting duplicated functions. The number of cofactors, μ, is called *column multiplicity.*

Definition 3. Given the partitioning of X into two disjoint subsets (X_1, X_2), called the *bound set* and the *free set*, respectively, Ashenhurst-Curtis decomposition of f is:

$$f(X) = g(h_1(X_1), h_2(X_1),\ldots, h_u(X_1), X_2).$$

Lemma 1. [2][6] The decomposition of f with functions $h_1(X_1)$, $h_2(X_1),\ldots, h_u(X_1)$ exists iff the number μ of different cofactors of f w.r.t. X_1 satisfies $\lceil \log_2\mu \rceil \le u$.

The BDD representation of functions is convenient for the computation of decompositions because the set of cofactor of F is found by detecting the nodes in the BDD pointed by the nodes above the cut separating X_1 from X_2 in the variable order [8][12].

Definition 4. [9] The *width* of the BDD at level k is defined as the column multiplicity of f with the bound set composed of variables above level k.

Definition 5. The *width profile* of the BDD is the ordered set of integers representing the width of the BDD at levels, starting from the topmost level 0 to the level of constant nodes.

The width profile of the BDD can be efficiently computed by one traversal of the BDD, visiting all the BDD nodes exactly once. The complexity of this algorithm is $O(N)$, while the complexity of the algorithm proposed in [9] is $O(n*N)$, where n is the number of variables and N is the number of nodes in the BDD.

During variable reordering the width of the BDD can be updated by modifying the width profile on the level where two adjacent variables are swapped.

Definition 6. Given the function f and the limit μ on the width of the BDD of f, the *OR-decomposition of* f *with the limited width* is:

$$f(X) = f_1(X) \vee f_2(X) \vee \ldots \vee f_m(X),$$

where the BDDs of $f_i(X)$ have the width no more than μ.

5.2. Decomposition Algorithm

In this subsection, we discuss the OR-decomposition algorithm as applied to the BDD. This algorithm works on a multi-output function represented by an SBDD+, an MTBDD+, or an ECFN. If the multi-output function is represented by an MTBDD, the decomposition is performed using the MTBDD+. In this case, the auxiliary variables are not considered as the input variables of the function.

The pseudo-code of the OR-decomposition algorithm is shown in Fig. 6. The algorithm is iterative. In each iteration, it extracts a dense subset of the BDD paths leading to the terminal node 1, in such a way that the width of this subset does not exceed the limit. The BDD minimization technique is the original Coudert's restrict algorithm [5] implemented in the CUDD package [17].

```
bdd_array OR_Decomposition( bdd F, int Limit )
{   bdd_array Result;
    bdd DontCare, S;
    DontCare = 0;
    while ( Width(F) > Limit )  {
         S = FindDenseSubsetOfPath( F );
         AddToArray( Result, S );
         DontCare = DontCare ∨ S;
         F = BddMinimize( F, DontCare );
    }
    return Result;
}
```

Figure 6. Pseudo-code of BDD decomposition algorithm.

The AND-decomposition can be performed similarly, by applying the OR-decomposition to the complement of the function. In the case, the final result is selected to be the best of the two decompositions.

Example 2. Consider a 1-bit adder introduced in Example 1 and its MTBDD+ shown in Fig. 5 (right). The maximum width of the MTBDD+ on the level of variable z is 3. The cascade with one rail requires that the width of the MTBDD+ would not exceed 2. Therefore, decomposition should be applied.

As a result of OR-decomposition, function $f_1 = abz$ is extracted. This function contains one path to the terminal node 1, and has the maximum width 2. The remaining function, $f_2 = (a \oplus b)\,\bar{z}$, also has the maximum width 2. The LUT cascade for f can be implemented as OR of LUT cascades for f_1 and f_2,

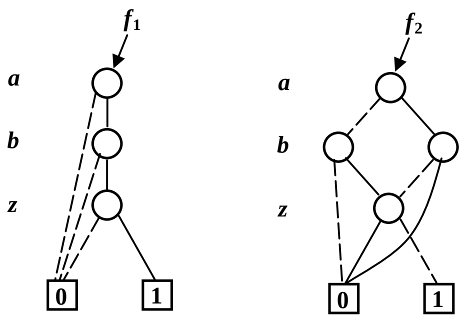

Figure 7. OR-decomposition of MTBDD+ for 1-bit adder.

Example 3. The LUT for f_2 is shown in Fig. 8. LUT-1 and LUT-2 of the cascade correspond to the upper and lower variables in the BDD of f_2 separated by the continuous line.

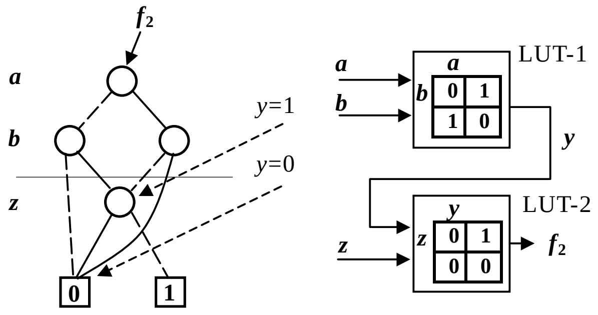

Figure 8. LUT cascade for function f_2.

6. Experimental Results

The OR-decomposition algorithm is implemented in C using BDD package CUDD Release 2.3.1 [17]. The algorithm is tested on benchmarks used in [10][15][16]. For each benchmark, the LUT cascade synthesis is performed using the ECFN for all outputs and the set of MTBDDs derived for the groups of outputs.

The LUT cascade parameters are selected differently in the two runs of synthesis. For the ECFN, we use 15-inputs LUTs with 14 rails because of the need to implement potentially wide BDDs. For the MTBDDs, we use much smaller LUTs (13 inputs, 8 rails), to show that out algorithm can fit large functions into cascades with the limited rails. In both runs of synthesis, if the width of the benchmark allowed for fewer rails than the given limit, the spare LUT inputs were used for additional side variables.

The grouping of outputs for synthesis with the MTBDDs is performed using a simple greedy algorithm. The group size is defined by the user on the command line. A new group is started with the output that has the largest support among the remaining outputs. Other outputs are added iteratively in such a way that each new output minimizes the increase in the support of the group after including the given output.

The experimental results are shown in Table 1. The following notations are accepted in the table. The benchmarks are described by listing their names, the number of inputs (“Ins”) and the number of outputs (“Outs”). The parameters of the ECFN are the number of BDD nodes (“nodes”) and the average width (“width”). The synthesis results for the ECFN contain the number of LUTs (“LUTs”), the number of cells (“cells”), and the total amount of memory in all LUTs, measured in megabytes (“mem”).

The grouping of outputs is described in Table 1 by showing the group size (“size”), the number of groups (“grs”) and the maximum support size of a group (“supp”). If the number of outputs in the given benchmark is not divisible by the group size evenly, the last group contains fewer outputs than the group size.

The DD parameters for the MTBDDs are similar to those for the ECFN, the only difference being that the *largest* number of nodes and average width among all groups are shown in the corresponding columns of Table 1. Synthesis results for MTBDDs contain some additional columns: the number of different branches produced by the decomposition (“bran”), and the length of the longest branch of the cascade measured in terms of cells (“max”). When the number of different branches produced by decomposition ("bran") is greater than the number of groups ("grs"), OR gate(s) must be used to combine the sub-functions.

The total runtime of the decomposition algorithm used in synthesis with MTBDDs for all benchmarks listed in Table 1 is about 5 seconds on an Intel 2GHz Pentium 4 CPU with 256Mb RAM under Microsoft Windows 2000. This time does not include the time of reading the benchmarks from BLIF files and reordering the variables to reduce the BDD width.

The results in Table 1 show that synthesis with MTBDDs on average increases memory requirements by approximately 9% compared to synthesis with the ECFN. However, for some large benchmarks (for example, *c880.blif* and *c2670.blif*) the memory was reduced about two times.

The main advantage of the new synthesis flow is the speed of resulting LUT cascades. In the case of the ECFN, each output is evaluated independently, and each evaluation takes as many memory lookups as there are cells in the LUT cascade. In the case of the MTBDDs, the evaluation is performed simultaneously for all outputs and one evaluation time takes as many memory lookups as there are cells in the longest branch of the cascade (the column “max” in the synthesis results with MTBDDs).

Assuming that one memory lookup takes the same amount of time for LUTs with any number of inputs, the experimental results show that the overall speed up due to shorter LUT cascades and simultaneous evaluation of outputs is about 100 times.

7. Conclusions

The LUT cascade provides an alternative way to realize multi-output combinational logic functions, occupying an intermediate position between hardware implementation (FPGAs, CPLDs) and software simulation (branching programs).

Previous approaches to LUT cascade synthesis assumed the availability of arbitrarily large LUTs to implement complex multi-output functions. In this paper, we presented an algorithm to synthesize LUT cascades with the limit on the number of rails.

The proposed algorithm allows for additional flexibility in implementing large designs and leads to significantly faster circuits at the cost of insignificant increase in memory consumption. Experimental results show that the algorithm gives favorable results on benchmark functions.

8. Acknowledgements

The authors gratefully acknowledge the support from Kyushu Institute of Technology under the 75th Commemoration Fund Program for Foreign Researchers. This research is partially supported by the Aid for Scientific Research from the Japan Society for the Promotion of Science (JSPS), and a grant from the Takeda Foundation.

References

[1] P. Ashar and S. Malik, "Fast functional simulation using branching programs", Proc. International Conference on Computer-Aided Design (ICCAD 1995), pp. 408-412, Oct. 1995.

[2] R. L. Ashenhurst, "The decomposition of switching functions". In Proceedings of International Symposium on the Theory of Switching, pp. 74-116, April 1957.

[3] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation". IEEE Trans. Computers, C-35 (8), pp. 677-691, August 1986.

[4] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping", Proc. Design Automation Conference, pp. 54-60, June 1993.

[5] O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution", in Automatic Verification Methods for Finite State Systems. Berlin, Germany: Springer-Verlag, 1989, pp. 365-373.

[6] H. A. Curtis, A New Approach to the Design of Switching Circuits, D. Van Nostrand Co., Princeton, NJ, 1962.

[7] H. Gouji, T. Sasao, and M. Matsuura, "On a method to reduce the number of LUTs in LUT cascades". Technical Report of IEICE, VLD2001-99, Nov. 2001 (in Japanese).

[8] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula, "EVBDD-based algorithm for integer linear programming, spectral transformations, and functional decomposition", IEEE Trans. CAD, Vol. 13, No. 8, pp. 959-975, Aug. 1994.

[9] S. Minato, "Minimum-width method of variable ordering for binary decision diagrams", IEICE Trans. Fundamentals, Vol. E75-A, No. 3, pp. 392-399, Mar 1992.

[10] A. Mishchenko and T. Sasao, "Encoding of Boolean functions and its application to LUT cascade synthesis," International Workshop on Logic and Synthesis (IWLS 2002), New Orleans, Louisiana, June 4-7, 2002, pp.115-120.

[11] S. Nagayama, T. Sasao, Y. Iguchi and M. Matsuura, "Representations of logic functions using QRMDDs," 32th International Symposium on Multiple-Valued Logic (ISMVL 2002), Boston, U.S.A, May 22-24, 2002, pp.261-267.

[12] T. Sasao, "FPGA design by generalized functional decomposition". In T. Sasao (ed.), Logic Synthesis and Optimization, Kluwer Academic Publishers, 1993.

[13] T. Sasao, "Compact SOP representations for multiple-output functions: An encoding method using multiple-valued logic," 31th International Symposium on Multiple-Valued Logic, Warsaw, Poland, May 22-24, 2001, pp.207-212.

[14] T. Sasao, "Design methods for multi-rail cascades," (invited paper) International Workshop on Boolean Problems (IWBP 2002), Freiberg, Germany, Sept. 19-20, 2002, pp. 123-132.

[15] T. Sasao, Y. Iguchi and M. Matsuura, "Comparison of decision diagrams for multiple-output logic functions," International Workshop on Logic and Synthesis (IWLS 2002), New Orleans, Louisiana, June 4-7, 2002, pp.379-384.

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

[17] F. Somenzi. *CUDD package, Release 2.3.1.* http://vlsi.Colorado.EDU/~fabio/CUDD/cuddIntro.html

Table 1. Experimental results for LUT cascade synthesis using the ECNF and the MTBDDs with output grouping.

Benchmark			Synthesis using ECFN, u=14, k=15					Synthesis using MTBDD, u=8, k=13									
Name	Ins	Outs	DD parameters		Synthesis results			Grouping			DD parameters		Synthesis results				
			nodes	width	LUTs	cells	mem,Mb	size	grs	supp	nodes	width	bran	LUTs	cells	max	mem,Mb
c432	36	7	1081	48	20	4	0.078	8	1	36	1788	94	1	45	6	6	0.044
c499	41	32	27174	596	63	8	0.246	4	8	41	3566	84	10	367	59	6	0.358
c880	60	26	9768	256	64	9	0.250	1	26	45	1216	19	26	179	58	6	0.175
c1908	33	25	9266	265	35	5	0.137	4	7	33	4079	64	9	224	38	5	0.219
c2670	233	140	6635	108	204	30	0.797	2	70	119	916	8	70	355	135	12	0.347
c3540	50	22	39222	1151	104	11	0.406	1	22	50	11683	126	51	1380	263	8	1.348
c5315	178	123	3744	98	154	23	0.602	2	62	67	1386	16	62	1062	282	9	1.037
c7552	207	108	11139	169	219	29	0.855	4	27	194	2292	18	27	1077	260	18	1.052
apex3	54	50	1247	42	35	7	0.137	8	7	45	378	20	7	140	30	6	0.137
apex7	49	37	533	28	27	6	0.105	8	5	26	876	31	5	76	13	3	0.074
b9	41	21	224	14	15	4	0.059	8	3	20	244	14	3	30	6	2	0.029
dalu	75	16	1179	82	57	9	0.223	4	4	53	1708	51	6	163	32	7	0.159
des	256	245	4024	129	244	34	0.953	4	62	22	471	12	62	454	126	3	0.443
duke2	22	29	464	25	10	3	0.039	8	4	19	190	14	4	39	8	2	0.038
e64	65	65	757	17	27	7	0.105	8	9	65	92	5	9	118	45	6	0.115
ex4	128	28	597	20	39	9	0.152	8	4	34	486	16	4	38	10	4	0.037
k2	45	45	1815	57	30	6	0.117	8	6	44	545	24	6	150	29	6	0.146
rot	135	107	8090	164	128	18	0.500	4	27	60	3533	28	28	500	103	8	0.488
spla	16	46	579	34	7	2	0.027	8	6	16	196	12	6	55	12	2	0.054
Total					1482	224	5.788		360				396	6452	1515	119	6.300
Ratio						100.0	100.0									53.1	108.8