Enumeration of Irredundant Circuit Structures

Alan Mishchenko

Department of EECS, University of California, Berkeley

alanmi@eecs.berkeley.edu

Abstract

A new approach to Boolean decomposition and matching is proposed. It uses enumeration of all support-reducing decompositions of Boolean functions up to 16 inputs. The approach is implemented in a new framework that compactly stores multiple circuit structures. The method makes use of pre-computations performed offline*, before the framework is started by the calling application. As a result, the runtime of the* online *computations is substantially reduced. For example, matching Boolean functions against an interconnected LUT structure during technology mapping is reduced to the extent that it no longer dominates the runtime of the mapper. Experimental results indicate that this work has promising applications in CAD tools for both FPGAs and standard cells.*

1. Introduction

Logic synthesis plays important role in hardware design and verification. The objective of logic synthesis is to come up with an optimal structural (gate-level) representation for a Boolean function originally represented as truth table, BDD, SOP, or a suboptimal logic structure.

Boolean decomposition is a technique that breaks down a given function into several smaller functions. Recursively applying decomposition can result in a network of logic nodes. To a large extent, past research concentrated on developing decomposition algorithms and applying them to practical problems [5][8][11][19]. For example, support-reducing decomposition [11] has been shown to be useful in mapping and post-mapping resynthesis [12].

The drawbacks of Boolean decomposition are two-fold: slow runtime and suboptimal quality. The latter is often because most of the decomposition algorithms are heuristic and therefore suboptimal. Attempts to develop exact algorithms require enumerating all feasible decompositions, which is often impractical due to prohibitive runtime.

The present work addresses both of these drawbacks.

The first contribution is a software package for harvesting small practical functions (SPFs) appearing in the benchmark circuits and caching the results of exhaustive Boolean decomposition applied to these functions. Two unexpected findings are: (1) The number of unique Boolean functions appearing in typical designs is manageable. A practical subset of them can be harvested in a few hours on a modern computer and stored in less than 1GB of memory. (2) The complete set of decompositions for practical functions is also not prohibitively large. Similar to functions themselves, these decompositions can be pre-computed and stored given the same runtime and memory requirements. The use of pre-computation in CAD algorithms is not new. One example is minimum Steiner tree computation [9].

The second contribution is a new algorithm for Boolean matching into LUT structures [23], which improves the quality and reduces the runtime of this important computation, which found several applications in synthesis and mapping for FPGAs. In particular, it can be used to improve the quality of the traditional LUT mapping.

A similar framework was proposed in the previous work on harvesting circuit structures [26]. The key difference of the current work is that it exhaustively pre-computes circuit structures, instead of collecting an incomplete subset of structures appearing in available gate-level benchmarks.

The rest of the paper is organized as follows. Section 2 contains necessary background. Section 3 describes exhaustive enumeration of irredundant decompositions and circuit structures. Section 4 outlines the DSD manager used in this work. Section 5 describes an application to LUT structure mapping. Experimental results are given in Section 6, while Section 7 concludes the paper.

2. Background

2.1 Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to logic gates and edges corresponding to wires connecting the nodes. It is assumed that each node has a unique integer called *node ID*.

A node *n* has zero or more fanins, i.e. nodes driving *n*, and zero or more fanouts, i.e. nodes driven by *n*. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A fanin (fanout) cone of node *n* is a subset of nodes of the network, reachable through the fanin (fanout) edges of the node.

2.2 And-Inverter Graph

And-Inverter Graph (AIG) is a Boolean network whose nodes can be classified as follows:

- One constant 0 node.
- Combinational inputs (primary inputs, flop outputs).
- Internal two-input AND nodes.
- Combinational outputs (primary outputs, flop inputs).

The fanins of internal AND nodes and combinational outputs can be complemented. The complemented attribute is represented as a bit mark on a fanin edge, rather than a separate single-input node.

Due to their compactness and homogeneity, AIGs have become a de-facto standard for representing circuits in technology mappers.

2.3 Structural cuts

A cut C of a node n is a set of nodes of the network, called *leaves* of the cut, such that each path from a PI to n passes through at least one leaf. Node n is called the root of cut C. The cut *size* is the number of its leaves. A trivial cut is the node itself. A cut is *K-feasible* if the number of leaves does not exceed K. A local function of an AIG node n, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in n and expressed in terms of the leaves, x, of a cut of n.

Cut enumeration [22] is a technique used by a cut-based technology mapper to perform cut computation using dynamic programming, starting from PIs and ending at POs.

2.4 Boolean functions

Let $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$, be a completely specified Boolean function, or *function* for short. The *support* of *f, supp*(f), is the set of variables X influencing the output value of f. The support size is denoted by $|X|$.

In this paper, completely-specified Boolean functions whose support size does not exceed 16 are called *small practical functions* (SPFs).

Two functions are NPN-equivalent if one of them can be obtained from the other by negation (N) and permutation (P) of the inputs and outputs. Consider the set of all Boolean functions derived from a given function F by a sequence of these transforms. These functions constitute the *NPN class* of function F. The *NPN canonical form* of function F is one function belonging to its NPN class, also called the *representative* of this class. Selection of the representative is algorithm-specific. For example, in some cases, the representation is the function whose truth table has minimum (or maximum) integer value among all the truth tables of functions belonging to the NPN class.

2.5 Boolean decomposition

For a function $f(X)$ and a subset of its support, X_1, the set of distinct *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 $2^{|X_1|}$ assignments of X_1 into $f(X)$ and eliminating duplicated functions. The number of distinct cofactors, μ, is the *column multiplicity* of f w.r.t. X_1.

Given a partition of X into two disjoint subsets, X_1 and X_2, Ashenhurst-Curtis decomposition [1][10] of $f(X)$ is:

$$f(X) = h(g_1(X_1), g_2(X_1), \ldots, g_k(X_1), X_2).$$

where subsets X_1 and X_2 are called the *bound set* and the *free set*, respectively. Functions $g_i(X_1)$, $1 \le i \le k$, are the *decomposition functions*. Function $h(G, X_2)$ is the *composition function*.

The decomposition of $f(X)$ with k functions $g_1(X_1)$, $g_2(X_1)$, …, $g_k(X_1)$ exists if and only if (iff) $\lceil log_2 \mu \rceil \le k \le n$, where μ is the column multiplicity of f w.r.t. X_1, and $n = |X_1|$. The bound set X_1 leads to an n-to-k *support-reducing decomposition* (SRD) [11] if k satisfies $\lceil log_2 \mu \rceil \le k < n$.

2.6 Disjoint-support decomposition

Disjoint-support decomposition (DSD) [1][3][15][24] is a special-case of Boolean decomposition when there is only one decomposition function $g(X_1)$. If DSD exists, it can be recursively applied to the decomposition/composition functions, resulting in a tree of nodes with non-overlapping supports and inverters as optional complemented attributes on the edges. This tree is called the *DSD structure* of a function. DSD is *full* if all logic nodes are elementary gates (AND, XOR, MUX). This definition is motivated by the fact that the elementary gates appear often in the designs and therefore DSD structures composed of them capture a practical subset of functions.

If a full DSD exists for a given function, it is unique up to a permutation of inputs and positions of inverters on the edges [1]. For example, $F = ab + cd$ has full DSD containing three ANDs and three inverters; F =!(!(ab) !(cd)). If a full DSD does not exist, the function can be represented by a decomposition tree composed of some elementary gates and one or more nodes not decomposable by DSD, called *prime* nodes [3], or non-DSD nodes. In the trivial case when there is no DSD, the function is represented by a prime node. For example, the *gamble function* ("all or nothing"), $F = abcdef + !a!b!c!d!e!f$, has no DSD but can be decomposed using a more general Boolean decomposition with one shared variable.

In this paper, if a function has some non-trivial DSD, it is called *DSD-able*. Alternatively, a function whose DSD tree contains only one prime node, is called *non-DSD-able*. If a function can be decomposed by a support-reducing Boolean decomposition, it is called *SRD-able*. Alternatively, if a function has no support reducing decomposition, it is called *non-SRD-able*. Note that every DSD-able function is also SRD-able. However, not every SRD-able function is DSD-able. For example, the gamble function listed above is SRD-able with any non-trivial bound set with one shared variable, but it is not DSD-able with any bound set.

2.7 Non-disjoint-support decomposition

This section considers the case when DSD does not exist. That is, there is no SRD with k=0. Given a bound set that allows for a SRD with k decomposition functions, k > 0, there are many possible sets of composition/decomposition functions that can be used to decompose the function. They represent different strict and non-strict encodings [16] of the μ cofactors of the original function f w.r.t. variables X_1.

SRD with k > 1 can often be encoded in such a way that there is only one non-trivial decomposition function $g_1(X_1)$, while other decomposition functions $g_2(X_1)$, …, $g_k(X_1)$, are represented by functions, each of which is equal to an elementary variable. The advantage of such functions is that in hardware they are implemented using interconnect, without dedicated gates or LUTs. In this case, decomposition has the following form

$$f(X) = h(g(X_1, X_3), X_3, X_2),$$

where the subset X_3 is called the *shared set*. The shared set variables are included in both the bound set and the free set. This decomposition is known as *non-disjoint-support decomposition* (NDSD), in contrast to disjoint-support decomposition (DSD) discussed above.

NDSD with a bound set, a free set, and a shared set is called *irredundant*, if the shared set is minimal. In other words, given X_1, X_2, and X_3, it is not possible to remove any variable either from X_1 and X_3, or from X_2 and X_3, and have NDSD with these variable sets.

3. Enumerating decompositions

This section describes an exhaustive enumeration of all irredundant non-disjoint-support decompositions (NDSDs) of the given Boolean function represented as a truth table.

The algorithm iterates over different bound sets and shared sets, while monotonically increasing their size. This allows for finding irredundant decompositions first. To this end, each decomposition derived is checked against the available ones and saved only if it is irredundant. The truth-table-based implementation of this algorithm is efficient because performs all the checks in-place, without duplicating or swapping binary data of the truth table. If decomposition exists, the set of all feasible decomposition and composition functions can also be efficiently derived.

It should be noted that computation of irredundant decompositions is only applied to representatives of each NPN class of non-DSD-able functions. All other non-DSD-able functions can be decomposed by mapping them into the representative of the NPN class and reusing the data available for the representative.

The set of functionally different SPFs encountered as functions of structural cuts during cut enumeration is relatively small. Typically it is does not exceed 10% of the number of structural cuts in each design. The percentage is even smaller when a family of designs are considered, whose logic often has substantial similarities. The number of unique NPN classes of non-DSD-able prime functions appearing in the DSD structures representing a set of SPFs is also quite small. It typically constitutes 10% of the number of DSD structures stored in the DSD manager. As a result, the number of different non-DSD-able functions, to which the exhaustive enumeration of irredundant decompositions is applied, is typically less than 1% of the number of cuts enumerated by the technology mapper.

These decompositions are pre-computed by running technology mapping for a given design or suite of designs, next saving the resulting DSD manager into a file, and later reusing it for mapping of the same designs, possibly with different settings of the mapper. If new SPFs are encountered during this mapping, they can be dynamically added to the DSD manager or skipped. The former may affect the runtime, although the slow-down is typically negligible. The latter may affect the quality of results, although the degradation is also often negligible provided that the number of new SPFs is small (say, 1-3%). These observations are confirmed by experimental results.

An interesting observation regarding the use of pre-computed decompositions, is that each NDSDs with k variables in the shared can be realized using exactly $2^{(2^k)}-1$ different pairs of NPN classes of decomposition and composition functions (g and h in the above formulas). This is because there are 2^k of pairs of cofactors of functions g and h with k shared variables. Each cofactor pair can be phase-assigned independently. In other words, we can always borrow an inverter from a cofactor of h and add it to the corresponding cofactor of g, or vice versa. If we remove one phase assignment leading to the same NPN class of functions g and h, there is $(2^k)-1$ non-NPN-equivalent phase assignments to be chosen independently, resulting in precisely $2^{((2^k)-1)}$ pairs of NPN classes.

4. DSD manager

DSD manager is a software package for storing and manipulating SPFs represented by their DSD structures.

Below is a short summary of the salient features of the DSD manager. Additional details can be found in Section 4 of previous work [20].

A DSD manager is similar to a BDD manager [7] in that it represents a given function it terms its canonical graph composed of elementary gates (AND, XOR, MUX) and prime nodes. Since DSD is canonical, the DSD manager always first performs DSD, if it exists. The remaining non-DSD-able functions become prime nodes. Each prime node is annotated with the representative of its NPN class, which is, in turn, annotated with the complete set of irredundant NDSDs, computed as shown in Section 3.

5. Applications to LUT mapping

K-LUT is a programmable device that can implement any Boolean function whose support size does not exceed K. Mapping into K-LUTs attempts to cover the given netlist with K-input structural cuts while minimizing delay and area of the mapping. Each cut can be implemented as one K-LUT in hardware implementation.

Mapping into LUT structures [23] attempts to map the netlist into M-input logic cells composed of two or more K-LUTs. For example, a 7-input cell called “44” [23] is composed of 2 4-LUTs, or a 16-input cell called “666” is composed of 3 6-LUTs arranged as a cluster (two LUTs feeding into the third one), or as a cascade (the first LUT feeding into the second one, feeding into the third one).

The main task when mapping into LUT structures is to check whether a given M-input Boolean function can be implemented (matched) with a given LUT structure. The heuristic matching algorithm proposed in [23] is incomplete, that is, in some cases, an existing match is not found. The algorithm offers a trade-off between quality and runtime. However, in many cases, a better runtime is desired, especially when mapping into LUT structures whose support size exceeds 10, for example, a 11-input structure composed of two 6-LUTs (called “66”).

To address this problem, we developed a matching algorithm, which utilizes the DSD structure of a function stored. Recall that this structure is annotated with the set of all support-reducing decompositions of each prime node, if such nodes are present in the DSD structure.

The algorithm is illustrated using an example below.

Suppose a 6-input function Y is to be mapped into 7-input LUT structure “44”. Suppose Y has DSD structure given by the formula $Y = a \,\&\, Z(b, c, d, e, f)$, where function Z is a prime node. Next, consider pre-computed support-reducing decompositions of Z of the type: $Z = H(G(X_1, X_3), X_3, X_2)$. Since “44” has 7 inputs and F depends on 6 variables, there can only be one shared variable, that is, $|X_3| = 1$. It is

possible to conclude that Y has a match with "44" if Z has a decomposition with $|X_1| = 4$ and $|X_2| = 3$. Suppose this decomposition is $Z = H(\ G(b, c, d, e), b, f\ )$. In the final realization of F, the 4-input function $g(b, c, d, e)$ is realized by the first 4-LUT, while the leftover 4-input function a & $H(g, b, f)$ is realized by the second 4-LUT.

Upon a closer investigation, the above DSD-based matching algorithm for LUT structures is incomplete, that is, it does not always detect a match when a match exists.

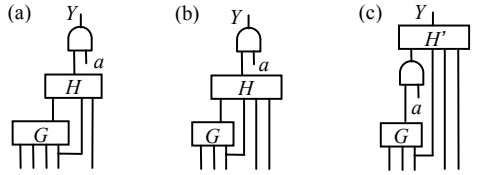

Figure 1: Illustration of a matching situation.

Consider Figure 1. If function Z is decomposable as shown in Figure 1(a), the match with "44" will be found. Suppose that function Z cannot be decomposed as shown in Figure 1(a) but can be decomposed as shown in Figure 1(b). In this case, the proposed algorithm concludes that the match with "44" does not exist. However, in a rare case shown in Figure 1(c) it may be possible to move the AND gate over a non-decomposable function H resulting in another non-decomposable function H'. In this case, the match with "44" exists but is not found by the proposed algorithm. Our experiments show that such cases are very rare. In particular, among SPFs not decomposable as shown in Figure 1(a) but decomposable as shown in Figure 1(b), less than 1% can be decomposed as shown in Figure 1(c).

6. Experimental results

The proposed framework is implemented in ABC [2]. The DSD manager is maintained as a data-structure separate from the LUT mapper *if* [18]. The mapper is used to perform LUT structure mapping using pre-computed decompositions. If a new function is encountered, its decompositions are computed on-the-fly and added to the DSD manager. Several commands are added to ABC to perform operations on the DSD manager, such as serialization (*dsd_load, dsd_save*), merging of two managers (*dsd_merge*), and printing statistics (*dsd_ps*).

The following four experiments are reported.

6.1 DSD structures

This experiment profiles the complete set of DSD structures computed for functions of structural cuts computed for all circuits from MCNC, ISCAS, and ITC benchmarks sets. To this end, all circuits were mapped into 12-LUTs, while keeping at most 16 priority cuts at each node [18]. (Note that when mapping into K-LUTs, some cuts have size smaller than K.)

The resulting DSD manager has 3.5M different DSD structures with 1.4M prime nodes annotated with 0.5M different NPN classes of functions up to 12 variables. The binary file containing the DSD manager has size 177MB. The zipped archive has size 42MB. It is available for downloading from [27]. Harvesting DSD structures took about 90 minutes on a modern workstation. The runtime included structural cut enumeration, truth table computation, DSD computation from truth tables, computation of canonical forms of DSD structures, and exhaustive enumeration of support-reducing non-disjoint-support decompositions for each NPN class of prime nodes. The latter computation took about 40% of the total runtime.

Table 1 shows 20 most frequently occurring DSD structures having supports 6, 9, and 12 inputs. The table lists formulas and the occurrence counters. The notation is as follows: $!a$ denotes complementation NOT(a); (ab) denotes AND(a,b); [ab] denotes XOR(a,b), and <abc> denotes MUX(a, b, c) = ab + $!ac$. Prime functions (not shown in the table) are denoted by their hexadecimal truth table followed by the list of arguments. For example, the 6-input gamble function from Section 2.6 is represented as follows: 0000000180000000{a, b, c, d, e, f}.

6.2 Non-decomposable functions

In this experiment, we profile the number of different irredundant decompositions for non-DSD-able functions appearing in DSD structures of functions computed during mapping of benchmarks circuits.

The same DSD manager (described in Section 6.1) was used in this experiment. The total of 0.4M different NPN classes of functions up to 12 variables were considered.

Table 2 contains one row for DSD structures of each size listed in column "N". The total number of DSD structures of this size is given in column "Total". The following 12 columns ("0", "1-10", … "91-100", "More") list the percentages of functions among those listed in column "Total", which have the given number of decompositions. For example, there are exactly four NPN classes of non-DSD-decomposable functions of three variables and they have no SRDs. This is why 100% of three variable functions are listed in column "0". Finally, columns "Max" and "Ave" show the maximum and the average numbers of decompositions for each function in a row.

The table shows that the complete set of irredundant non-disjoint-support support-reducing decompositions is reasonable even for large value of N. For example, there are on average 37.7 decompositions for a typical 10-variable function. This set is relatively small and can be efficiently pre-computed and stored, as shown in Section 3.

6.3 Improvements to LUT structure mapping

This experiment considers the results of mapping into LUT structure "66" composed of two 6-LUTs. Two approaches are compared: the known heuristic algorithm based on partial enumeration of candidate bound sets [23] and the new algorithm introduced in this paper, which is based on exhaustive search over all bound sets. A set of ten industrial designs was used in this experiment. Functional equivalence of the original and final netlists was checked using an equivalence checker in ABC.

Table 3 shows the LUT count, the LUT level count, and the runtime, in seconds, for the two approaches. The right-hand side of the table contains two runtime columns. The first one shows the runtime of the mapper while DSD manager is being populated. The second one shows the

runtime of the mapper with the pre-computed DSD manager. The mapped circuits produced by these runs are identical. The former is slower because of decomposition and matching performed while mapping, whereas the latter relies on pre-computed results stored in the DSD manager.

When the proposed algorithm is used in a realistic FPGA synthesis flow, it will rely on pre-computation. In this case, the runtime will be close to that listed in the last column, which is about 3x faster than the previous work.

Careful examination of Table 3 shoes that, when the DSD manager is used, area for designs 02 and 04 has increased by about 0.2%. This happens due to the heuristic nature of area recovery during technology mapping.

6.4 Delay minimization in LUT mapping

This experiment shows that the depth of the 6-LUT mapping measured in terms of the number of LUT levels can be reduced by mapping into LUT structure "66" while considering the arrival times of the cut leaves. This approach is called *LUT balancing* (LUTB) because it is similar to SOP balancing (SOPB) [20] and Lazy Man's Synthesis (LMS) [26], except that in LUTB the delay is expressed in terms of LUT levels rather than AIG levels.

Table 4 shows the results obtained by applying four runs: (1) classical 6-LUT mapping by the mapper *if* [18]; (2) LUTB; (3) SOPB followed by LUTB; (4) LMS followed by LUTB. The same set of structural choices computed using command *dch -f* was used in all four runs.

Table 4 shows that AIG level optimization with SOPB and LMS helps reduce LUT level after LUTB. In particular, LUTB alone reduced the LUT level by 6%, compared to standard LUT mapping. Meanwhile, SOPB+LUTB and LMS+LUTB reduced it by 10.4% and 12.8%, respectively.

7. Conclusions

The paper introduces a framework for harvesting and utilizing Boolean functions appearing in hardware designs. The proposed exhaustive approach to Boolean decomposition and matching is made practical by the use of pre-computation: the runtime of the mapper based on matching functions of each cut against a library of standard cells or LUT structures is reduced several times, while at the same time improving the quality of results.

Current results show area reduction of about 1% and runtime reduction about 3x. Future developments will be to refine the implementation, adapt the framework to standard cell mapping and resynthesis, and apply the pre-computation of decompositions to improve technology-independent synthesis, which is an important component of both logic synthesis and formal verification tools.

Acknowledgements

This work is partly supported by SRC contract 1875.001 and NSA grant "Enhanced equivalence checking in crypto-analytic applications". We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Mentor Graphics, Microsemi, Real Intent, Synopsys, Tabula, and Verific for their continued support.

8. REFERENCES

[1] R. L. Ashenhurst, "The decomposition of switching functions". *Computation Lab*, Harvard University, 1959, Vol. 29, pp. 74-116.

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[3] V. Bertacco and M. Damiani, "Disjunctive decomposition of logic functions," *Proc. ICCAD '97*, pp. 78-82.

[4] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.

[5] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[6] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli, "Multilevel logic synthesis", *Proc. IEEE*, Vol. 78, Feb.1990.

[7] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comp*., Vol. C-35, No. 8 (August, 1986), pp. 677-691.

[8] M. R. Choudhury and K. Mohanram, "Bi-decomposition of large Boolean functions using blocking edge graphs". Proc. ICCAD'10, pp. 586-591.

[9] C. C. N. Chu, Y.-C. Wong, "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design", *IEEE TCAD'08*, Vol. 27(1), pp. 70-83.

[10] A. Curtis. *New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, NJ, 1962.

[11] V. N. Kravets. *Constructive Multi-Level Synthesis by Way of Functional Properties*. Ph. D. Thesis, University of Michigan, 2001.

[12] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization". *Proc. DAC'04*, pp. 438-441

[13] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps", *Proc. DAC'97*, pp. 263-268.

[14] H.-P. Lin, J.-H. R. Jiang, and R.-R. Lee. "Ashenhurst decomposition using SAT and interpolation," *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, S. Khatri and K. Gulati (Editors) , Springer 2011, pp. 67-85.

[15] Y. Matsunaga, "An exact and efficient algorithm for disjunctive decomposition", *Proc. SASIMI '98*, pp. 44-50.

[16] A. Mishchenko and T. Sasao, "Encoding of Boolean functions and its application to LUT cascade synthesis", *Proc. IWLS '02*, pp. 115-120. http://www.eecs.berkeley.edu/~alanmi/publications/2002/iwls02_enc.pdf

[17] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.

[18] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*, pp. 354-361.

[19] A. Mishchenko, R. K. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks", *Proc. ICCAD'08*, pp. 38-44.

[20] A. Mishchenko, R. Brayton, S. Jang, and V. Kravets, "Delay optimization using SOP balancing", *Proc. ICCAD'11*, pp. 375-382.

[21] A. Mishchenko and R. Brayton, "Faster logic manipulation for large designs", *Proc. IWLS'13*. http://www.eecs.berkeley.edu/~alanmi/publications/2013/iwls13_dsd.pdf

[22] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs", *Proc. FPGA '98*, pp. 35-42.

[23] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures", *Proc. DATE'12*.

[24] T. Sasao and M. Matsuura, "DECOMPOS: An integrated system for functional decomposition," *Proc. IWLS '98*, pp. 471-477.

[25] E. Sentovich, et al, "SIS: A system for sequential circuit synthesis", *Tech. Rep. UCB/ERI, M92/41*, ERL, Dept. of EECS, Univ. of California, Berkeley, 1992.

[26] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis", *Proc. ICCAD'12*.

[27] http://www.eecs.berkeley.edu/~alanmi/temp2/pub12_40417.zip

Table 1: The most frequently occurring NPN classes of DSD structures of 6-, 9-, and 12-input functions.

	6-input structures	Occurs	9-input structures	Occurs	12-input structures	Occurs
1	(a!(bc)!(d!(ef)))	4386	(abcdefg!(hi))	5511	(abcdefghij!(kl))	4092
2	(a!(b!(c!(d!(ef)))))	4128	(abcd!(ef)!(ghi))	5375	(abcdefghi!(j!(kl)))	2788
3	(ab!(c!(d!(ef))))	3727	(abcdef!(g!(hi)))	4901	(abcdefgh!(ij)!(kl))	2447
4	(a!(bc!(d!(ef))))	3503	(abcd!(ef)!(g!(hi)))	4625	(abcdefghi!(jkl))	2318
5	(a!(b!(cd!(ef))))	3075	(abcde!(fg)!(hi))	4588	(abcdefg!(hi)!(jkl))	2087
6	(!(ab)!(c!(d!(ef))))	2986	(ab!(cde)!(fghi))	4106	(abcdefghijkl)	2061
7	(ab!(cd!(ef)))	2788	(abc!(de)!(fghi))	3665	(abcdefg!(hijkl))	1880
8	(abc!(d!(ef)))	2727	(ab!(cd)!(efghi))	3575	(a!(bcdefghijkl))	1829
9	(a!(bc)!(def))	2477	(abcdef!(ghi))	3541	(a!(bc)!(defghijkl))	1736
10	(a!(b!(c!(def))))	2331	(!(abcd)!(efghi))	3437	(ab!(cdefghijkl))	1729
11	(a!(bcd!(ef)))	2231	(a!(bcd)!(efghi))	3411	(abcdefgh!(ijkl))	1697
12	(!(ab)!(cd!(ef)))	2226	(abc!(de)!(fg!(hi)))	2923	(abcdefg!(hi)!(j!(kl)))	1690
13	(ab!(cd)!(ef))	2077	(abcde!(f!(ghi)))	2911	(!(abcdef)!(ghijkl))	1564
14	(ab!(c!(def)))	2067	(a!(bc)!(defghi))	2863	(a!(!(bcdef)!(ghijkl)))	1543
15	(a!(b!(cd)!(ef)))	1807	(ab!(cd)!(ef)!(ghi))	2739	(abcdefghi<jkl>)	1489
16	(a!(!(bc)!(d!(ef))))	1713	(abcde!(fghi))	2737	(abcdef!(ghijkl))	1440
17	(a!(!(bc)!(def)))	1666	(ab!(cd)!(efg!(hi)))	2630	(abcde!(fghijkl))	1394
18	(a!(bc)[d(ef)])	1654	(abcde!(fg!(hi)))	2624	(abcd!(efghijkl))	1304
19	(a!(bc!(def)))	1633	(abc!(de)!(fg)!(hi))	2569	(!(ab)!(cdefghijkl))	1295
20	(ab[c(d!(ef))])	1608	(abcde!(f!(g!(hi))))	2553	(abcde!(fg)!(hijkl))	1259

Table 2: For each support size, the percentages of NPN classes having the given number of irredundant decompositions.

N	NPN	The number of different irredundant decompositions												Decs	Decs
	classes	0	1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	91-100	More	max	ave
0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0.0
1	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0.0
2	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0.0
3	4	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0.0
4	176	52.3	47.2	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	12	1.1
5	3438	12.8	81.7	5.2	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	50	4.1
6	17397	7.0	48.6	36.4	5.7	1.3	0.5	0.2	0.1	0.0	0.0	0.0	0.0	150	9.7
7	43926	2.1	32.0	31.8	19.6	7.3	3.0	1.5	0.8	0.5	0.4	0.2	0.7	392	17.9
8	78979	2.4	30.3	23.9	15.1	9.9	6.4	3.5	2.2	1.6	1.0	0.7	3.1	1000	25.9
9	104584	2.4	31.5	23.0	12.8	8.1	5.3	3.6	2.7	2.0	1.6	1.2	5.7	2214	32.3
10	106462	2.8	33.3	22.2	12.2	7.5	4.6	3.4	2.4	1.8	1.4	1.1	7.1	5454	37.7
11	83125	3.6	35.1	21.7	10.9	6.9	4.5	3.1	2.3	1.9	1.3	1.1	7.8	11132	42.8
12	41854	5.4	38.7	20.6	9.9	6.0	3.8	2.7	1.8	1.6	1.2	1.0	7.4	20144	47.3

Table 3: LUT structure mapping without DSD structures ([23]) and with DSD structures (this paper).

Design	Without DSD structures			With DSD structures			
	LUT	Level	Time, s	LUT	Level	Time, s	Time, s
01	33648	30	130.29	33212	30	180.27	32.33
02	19751	7	6.91	19777	7	3.37	2.36
03	28266	20	114.18	27859	20	123.18	52.49
04	40286	12	121.58	40332	12	55.22	32.36
05	47858	15	126.29	47016	15	91.29	32.99
06	95630	15	243.48	93901	15	123.09	60.70
07	32118	15	66.32	31564	15	73.01	16.67
08	33611	21	72.14	33083	21	32.68	16.63
09	34887	5	8.36	34835	5	9.68	3.60
10	13364	10	13.47	13218	9	67.70	4.68
Geomean	1.000	1.000	1.000	0.989	0.990	0.901	0.300

Table 4: The impact of DSD-based LUT balancing on minimizing LUT level during technology mapping.

Design	6-LUT mapping		LUTB		SOPB + LUTB		LMS + LUTB	
	LUT	Level	LUT	Level	LUT	Level	LUT	Level
01	32788	20	32483	18	31104	20	33047	17
02	19768	5	19818	5	20039	5	19956	5
03	27545	13	26716	13	27057	12	27081	12
04	39727	9	37644	9	39180	9	38906	8
05	46633	10	46225	9	46740	8	46754	8
06	93172	10	92238	9	92970	8	93270	8
07	31299	9	30929	8	30480	8	30811	8
08	36583	20	34730	19	36576	17	37334	17
09	33600	5	33455	5	33559	5	33565	5
10	13099	8	12943	7	13028	6	13011	6
Geomean	1.000	1.000	0.981	0.940	0.990	0.896	0.998	0.872