Finding All Simple Disjunctive Decompositions Using Irredundant Sum-of-Products Forms

Shin-ichi Minato
NTT Optical Network Systems Labs.
Kanagawa, Japan

Giovanni De Micheli
Stanford University
Stanford, CA

Abstract

Finding disjunctive decompositions is an important technique to realize compact logic networks. *Simple disjunctive decomposition* is a basic and useful concept, that extracts a single-output subblock function whose input variable set is disjunctive from the other part. This paper presents a method for finding simple disjunctive decompositions by generating irredundant sum-of-products forms and applying factorization. We prove that all simple disjunctive decompositions can be extracted in our method, namely, all possible decompositions are included in the factored logic networks. Experimental results show that our method can efficiently extract all the simple disjunctive decompositions of the large-scale functions. Our result clarifies the relationship between the functional decomposition method and the two-level logic factorization method.

1 Introduction

Functional decomposition is a fundamental theory of logic circuit design and has been studied for long time. *Simple disjunctive decomposition* is a basic and useful concept in this theory. This decomposition extracts a single-output subblock function whose input variable set is disjunctive from the other part. It is a special case of decomposition and not always possible for all Boolean functions. If we find a such decomposition for a given function, it must be a good choice for optimal design and we may proceed to the local optimization of each subblock. Thus, it is a good way to check simple disjunctive decompositions before applying other heuristic optimization methods. In addition, a simple disjunctive decomposition gives the minimum interconnection between the two subblocks, so it is also important for technology mapping and partitioning problems.

There are many studies on the method of finding simple disjunctive decompositions. At first, a classical method with a *decomposition chart* is presented[1, 10]. In last five years, more efficient way using a BDD-based implicit decomposition chart is discussed[6, 11, 12]. Recently, a further powerful algorithm[5] based on BDD traversal without a decomposition chart is proposed. Currently, functional decomposition technique is attracting a lot of interests from LSI CAD researchers in connection with FPGA architecture. However, there have not been any research of this topic with relation to the two-level logic minimization and factorization method, which is another popular design method widely used in commercial tools. In this paper, we propose a new functional decomposition algorithm that relates the two different logic design methods. Our result gives a theoretical backbone to the previous works on the functional decomposition technique.

In our decomposition algorithm, we first generate an irredundant sum-of-products form for given function, and then apply factorization. As a result of this synthesis process, we obtain a multi-level logic network that includes all simple disjunctive decompositions. We can easily find out all decompositions by traversing the logic network.

One of the key technique in our method is using Minato-Morreale algorithm[7] for generating irredundant sum-of-products forms. We show that this algorithm is not only fast but also having an important functional property, which is essential to perform disjunctive decomposition.

Our discussion and experimental results show that the two-level logic minimization and factorization method is strong enough to perform simple disjunctive decompositions. If the given function is dominated by simple disjunctive decompositions, our method produces a nearly optimal network and only local improvements remains in it. It shows a guideline to the choice of logic optimization strategies.

This paper is organized as follows. In Section 2, we describe the basic concepts. In Section 3 we show outline of our decomposition method and describe detailed algorithms. Experimental results are shown in Section 4, then we discuss the results. Section 5 concludes this paper.

2 Definitions and Basic Concepts

A *sum-of-products form* (*SOP* for short) is *irredundant*[1] if neither a literal nor a cube can be removed without changing the function. For example, $xyz + x\overline{y}$ is not irredundant, whereas $xz + x\overline{y}$ is irredundant. Irredundant SOPs are very compact in general, but not always minimum. They do not provide unique forms of Boolean functions.

If the function f can be represented as $f(X, Y) = g(h(X), Y)$, then f can be realized by the network shown in Fig. 1. We call it *simple disjunctive decomposition.* It

[1]Some people call this "prime and irredundant."

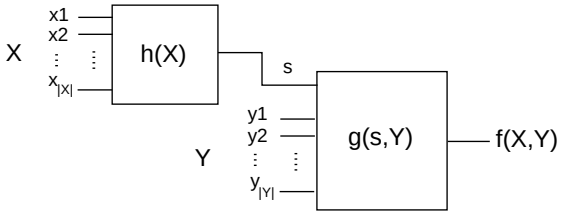


Figure 1: Simple disjunctive decomposition.

is called "simple" because h is a single-output function and "disjunctive" because X and Y have no common variables. We do not consider the trivial cases such that X consists of only one variable or all the support of f.

A simple disjunctive decomposition does not always exist in a given function, but if exists, it is considerably effective for logic optimization.

A function may have more than one simple disjunctive decompositions. They can be nesting. For example, the function $(a+b)(c+d)+ef$ has four decompositions as $X = \{a,b\}$, $\{c,d\}$, $\{e,f\}$, and $\{a,b,c,d\}$.

Multiple input logic operations (AND, OR, EXOR) may produce a number of symmetric decompositions. For example, $(a+b+c)$ can be decomposed as $(a+b)+c$, $(b+c)+a$ and $(a+c)+b$. n-input logic operation involves (2^n-n-2) sub-decompositions. In this paper, we handle such decompositions as one group, and extract the full-merged form to represent such a group. $(a+b+c)$ is the full-merged form of the above example.

Except sub-decompositions of the symmetric groups, two simple disjunctive decompositions never overlap each other. For example, $X = \{a,b,c\}$ and $\{b,c,d\}$ cannot coexist in the same function. Only nesting is possible. Therefore, an n-input function can have at most $(n-2)$ decompositions, excluding the symmetric sub-decompositions.

3 Decomposition Algorithms

Figure 2 shows the outline of our decomposition method. A function is given as a multi-level logic network. We first construct a BDD for the function, and generate a single irredundant SOP from the BDD. We then apply factorization to make a multi-level SOP network. Our method guarantees that the result of SOP network includes all simple disjunctive decompositions. We can easily find out all decompositions by traversing the network. Notice that the result of the networks are not unique for a given function if the different variable orderings are given, however, anyway the same set of simple disjunctive decompositions are found. Namely, the differences of the networks are only inside of the subblocks of the simple disjunctive decompositions.

Here we show why our method can extract all simple disjunctive decompositions.

3.1 Required Property in Generating Irredundant SOPs

Consider the function $f(X,Y)$ which has a simple disjunctive decomposition as $f(X,Y) = g(s,Y)$ and $s = h(X)$, as shown in Fig. 1. X and Y consist of

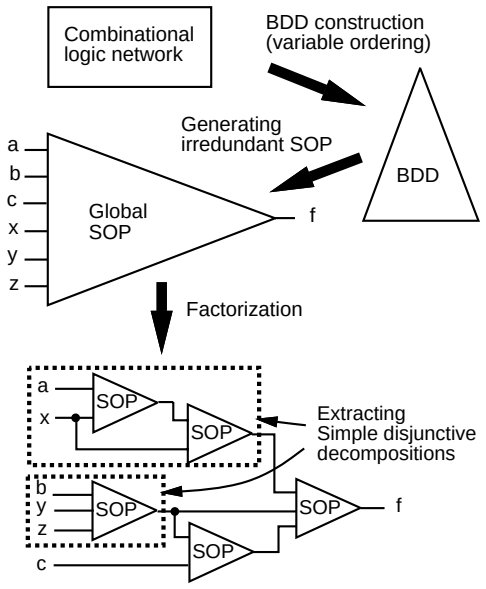


Figure 2: Outline of our method.

$\{x_1,\ldots,x_{|X|}\}$ and $\{y_1,\ldots,y_{|Y|}\}$, respectively. We call s the *substitution variable*.

Now we consider the relation between simple disjunctive decomposition and irredundant SOPs. Here $isop^f$ denotes the irredundant SOP for the function f. The $isop^g(s,Y)$ for the function $g(s,Y)$ can be factored into the three parts, as follows.

$$isop^g(s,Y) = s \cdot isop^g_P(Y) + \overline{s} \cdot isop^g_N(Y) + isop^g_D(Y), \quad (1)$$

where $isop^g_P(Y)$, $isop^g_N(Y)$, and $isop^g_D(Y)$ are also irredundant SOPs, not including variable s.

We then replace $s, \overline{s}$ with $isop^h(X), isop^{\overline{h}}(X)$, respectively.

$$isop^g(isop^h(X),Y) = isop^h(X) \cdot isop^g_P(Y) + isop^{\overline{h}}(X) \cdot isop^g_N(Y) + isop^g_D(Y), \quad (2)$$

Theorem 3.1 *An SOP obtained by expanding $isop^g(isop^h(X),Y)$ is an irredundant SOP for $f(X,Y)$.*

(Example)

$$isop^h(X) = x_1x_2 + \overline{x_1}x_3 + \overline{x_2}\ \overline{x_3},$$
$$isop^g(s,Y) = s\ y_1 + s\ y_2,$$
$$isop^g(isop^h(X),Y) = x_1x_2y_1 + \overline{x_1}x_3y_1 + \overline{x_2}\ \overline{x_3}y_1 + x_1x_2y_2 + \overline{x_1}x_3y_2 + \overline{x_2}\ \overline{x_3}y_2 \quad (\rightarrow \textit{irredundant SOP}) \quad (3)$$

Theorem 3.1 indicates that an irredundant SOP $isop^f(X,Y)$ can be obtained from $isop^g(s,Y)$, $isop^h(X)$, and $isop^{\overline{h}}(X)$. We then consider the reverse process to extract $isop^h(X)$, $isop^{\overline{h}}(X)$, and $g(s,Y)$ from $isop^f(X,Y)$.

For example, the literals y_1 and y_2 appear more than once in the last SOP, so they can be factored as:

$$isop^f(X,Y) = (x_1x_2 + \overline{x_1}x_3 + \overline{x_2}\ \overline{x_3})\ y_1$$

$+ (x_1x_2 + \overline{x_1}x_3 + \overline{x_2x_3})\ y_2$.

Here, the expression $(x_1x_2 + \overline{x_1}x_3 + \overline{x_2}\ \overline{x_3})$ reveals as a common cofactor of y_1 and y_2, and finally we can obtain $(x_1x_2 + \overline{x_1}x_3 + \overline{x_2}\ \overline{x_3})(y_1 + y_2)$. In this way, we can extract the decomposition function.

However, this is not always possible because the irredundant SOPs are not unique for a given function. For example, the same function can be represented as:

$$isop^f(X,Y) = x_1x_2y_1 + \overline{x_1}x_3y_1 + \overline{x_2}\ \overline{x_3}y_1 + x_1x_2y_2 + x_2x_3y_2 + \overline{x_1}\ \overline{x_2}y_2 + \overline{x_2}\ \overline{x_3}y_2,$$

and this is also an irredundant form. In this case, we cannot identify the decomposition function $h(X)$ because the cofactors of y_1 and y_2 are not identical.

From this observation, we can see that the following condition is required for extracting simple disjunctive decompositions.

Condition 3.1 (Uniformity of factors) *The decomposition function $h(X)$ must be represented uniquely as $isop^h(X)$ or $isop^{\overline{h}}(X)$ in $isop^f(X,Y)$, as shown in Eqn. 2. More exactly, for any assignment $\{0,1\}$ to each $y_j \in Y$, $isop^f(X,Y)$ must be reduced to one of the four expressions: 0, 1, $isop^h(X)$, or $isop^{\overline{h}}(X)$.*

If we can generate an irredundant SOP $isop^f(X,Y)$ that satisfies this condition, $isop^h(X)$ and $isop^{\overline{h}}(X)$ can be factored uniquely from $isop^f(X,Y)$.

3.2 Minato-Morreale Algorithm

Minimization or optimization of SOPs has extensively been studied for long time. Recently, Minato[7] developed a quite fast algorithm for generating an irredundant SOP directly from a given BDD. This algorithm is based on *recursive operator* shown by Morreale[9], and we call it *Minato-Morreale algorithm*. This algorithm is based on the recursive expansion respect to each input variable, and generates a unique SOP form under a fixed variable ordering. The detailed algorithm is described in Appendix.

The following theorem is the main contribution of this paper.

Theorem 3.2 *If a given function has a simple disjunctive decomposition as $f(X,Y) = g(h(X),Y)$, Minato-Morreale algorithm satisfies Condition 3.1 for any one fixed variable ordering.*

(Proof) *See Appendix.*

(Example) *Minato-Morreale algorithm generates the following different SOPs for the same function shown in Section 4.1 with the different variable orderings, and both SOPs satisfy the Condition 3.1.*

variable order $(x_1y_1x_2y_2x_3)$*:*

$$x_1y_1\overline{x_3} + \overline{x_1}y_1x_3 + y_1x_2x_3 + y_1\overline{x_2x_3} + x_1y_2\overline{x_3} + \overline{x_1}y_2x_3 + x_2y_2x_3 + \overline{x_2}y_2\overline{x_3}$$

variable order $(y_2x_2x_1y_1x_3)$*:*

$$x_2y_1x_3 + \overline{x_2}y_1\overline{x_3} + x_1y_1\overline{x_3} + \overline{x_1}y_1x_3 + y_2x_2x_3 + y_2\overline{x_2}\ \overline{x_3} + y_2x_1\overline{x_3} + y_2\overline{x_1}x_3.$$

Theorem 3.2 shows that a simple disjunctive decomposition can be extracted by factoring the SOP generated by Minato-Morreale algorithm. It is important that the theorem stands for any fixed variable ordering, so we do not have to determine the partition $\{X,Y\}$ beforehand. If more than one simple disjunctive decompositions are involved in a given function, the theorem stands for each decomposition, and all decompositions can be extracted by one sequence of factorings.

Irredundant SOPs can also be obtained by other algorithms, such as ESPRESSO[2] or simplify command in SIS[3], however, those algorithms do not satisfy the Condition 3.1. Thus they cannot be used for extracting simple disjunctive decompositions.

3.3 Factorization of SOPs

Next we confirm that all simple disjunctive decompositions can be extracted by one sequence of factorization. Here we consider the following three cases.

1. The subblock corresponding to $isop^h$ directly connects to the OR gate of the other part (Fig. 3(a)). In this case, no factoring is needed since $isop^h$ is just a subset of $isop^f$, and possibly it can be a subdecomposition of the symmetric group.

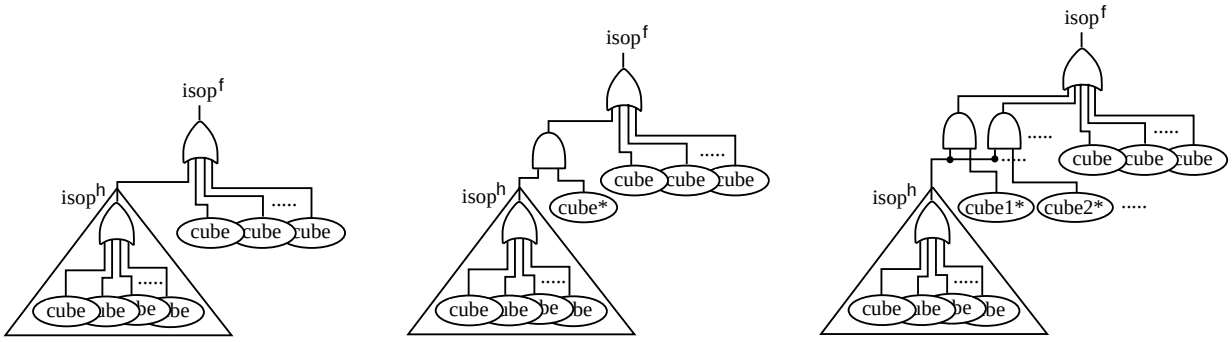
2. The subblock corresponding to $isop^h$ connects to only one AND gate of the other part (Fig. 3(b)). In this case, if $isop^h$ consists of only one cube, no factoring is needed similarly to Case 1.
   If $isop^h$ has multiple cubes, the cofactor cube (i.e. cube*) appears more than once in $isop^f$, along with each cube of $isop^h$. Therefore, $isop^h$ can be extracted by factoring a common literal set included in multiple cubes of $isop^f$.

3. The subblock corresponding to $isop^h$ fan-outs to more than one AND gates of the other part (Fig. 3(c)).
   In this case, if $isop^h$ consists of only one cube, it can be identified as a common literal set included in multiple cubes of $isop^f$.
   If $isop^h$ has multiple cubes, the each cofactor cube (i.e. cube1*, cube2*, ...) appears more than once in $isop^f$, along with each cube of $isop^h$. Since $isop^h$ is represented uniquely in the SOP, it can be merged into one block in the factorization process.

We discussed here $isop^h$ only, but $isop^{\overline{h}}$ can also be extracted similarly.

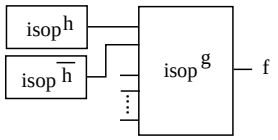
From this consideration, we can see that $isop^h$ and $isop^{\overline{h}}$ can eventually be extracted by repeating factorization of a common literal set. The important point here is that we do not need a information of partition $\{X,Y\}$ in factorization process. This implies that, if more than one simple disjunctive decompositions are involved in a given function, all of them are factored simultaneously in one sequence of factorizations. Notice that we need to traverse the SOP network to know where and how many simple disjunctive decompositions have been captured.

In the factorization process, ZBDD-based implicit SOP representation is also useful to perform factorization. Using a fast division algorithm[8], the computation time is almost linear with the size of ZBDDs, and it does not depend on the number of cubes or literals in SOPs.

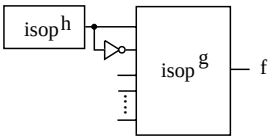


(a) Direct connection to OR gate. (b) Single fan-out to AND gate. (c) Multiple fan-outs to AND gates.

Figure 3: Classification of simple disjunctive decompositions.



(a) Before unification



(b) After unification.

Figure 4: Unification of complement subfunctions.

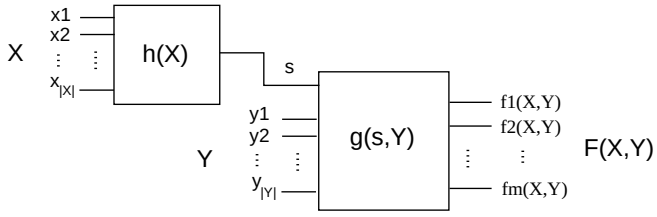


Figure 5: Simple disjunctive decomposition of a multi-output function.

3.4 Unification of complement subfunctions

Now we have $isop^h(X)$ and $isop^{\overline{h}}(X)$ extracted by factorization, as shown in Fig. 4(a). Obviously they share no common cubes each other. If we use a pure algebraic factorization, v and $\overline{v}$ are regarded as a different literals, and thus $isop^h(X)$ and $isop^{\overline{h}}(X)$ are factored separately. This result is not a "simple" decomposition yet. In order to combine them into one, we use the complement subfunctions in the factorization process, as Fig. 4(b). For example, $f = \overline{a}b + \overline{a}c + a\overline{b}\,\overline{c}$ can be factored using a complement subfunctions as $f = \overline{a}s + a\overline{s}$, $s = b + c$, $(\overline{s} = \overline{b}\,\overline{c})$. This is a really "simple" decomposition.

When the subfunction includes substitution variables, we should be careful to make a complement subfunction. Here we show a typical example.

$$f = as_3 + \overline{a}\,\overline{b}s_2 + \overline{a}\,\overline{s_1},$$
$$s_1 = c + d, \quad s_2 = \overline{c} + \overline{d}, \quad s_3 = bs_1 + \overline{s_2},$$

If we make just a complement of s_3, then $\overline{s_3} = \overline{b}s_2 + \overline{s_1}s_2$ cannot be used for factorization any more. However, $\overline{s_3}$ can also be represented as $\overline{b}s_2 + \overline{s_1}$, and then we find further factoring as $f = as_3 + \overline{a}\,\overline{s_3}$, which corresponds to a simple disjunctive decomposition of f. The problem is that Minato-Morreale algorithm guarantees unique forms of subfunctions if they represented with primary inputs. This property is broken when substitute variables are included.

To solve this problem, we re-construct a BDD of the complement subfunction with primary input variables only, and then generate irredundant SOPs by Minato-Morreale algorithm. After that we try factoring the SOP with all existing subfunctions in a fixed order. We maintain a list of all subfunctions and their complement which are extracted in the factorization process. In this way, we can keep the uniqueness of complement subfunctions.

Unification of complement subfunctions enables us to extract the decompositions with EXOR gates. If a function is decomposable with an EXOR gate, the sub-expressions such as $(h_1\overline{h_2} + \overline{h_1}h_2)$ or $(h_1h_2 + \overline{h_1}\,\overline{h_2})$ must be found in the factored SOP network. We detect those sub-expressions by pattern matching, and replace them with two-input EXOR gates. If the function involves multi-input EXOR functions, they must be factored as a cascade of two-input gates. For example, the three-bit parity function $f = xyz + x\overline{y}\,\overline{z} + \overline{x}y\overline{z} + \overline{x}\,\overline{y}z$ can be factored as $f = xs + \overline{x}\,\overline{s}$, $s = yz + \overline{y}\,\overline{z}$. We replace them as $f = \overline{x} \oplus s$, $s = \overline{y} \oplus z$, and merge them into $f = x \oplus y \oplus z$.

3.5 Extension to Multi-Output Functions

Our decomposition method can be extended to multi-output functions. We define simple disjunctive decomposition of multi-output function as shown in Fig. 5. If such a decomposition exists, each output function $f_k(X, Y)$ must have a same decomposition with $h(X)$, unless f_k is irrelevant to X. Thus, we perform Minato-Morreale algorithm for each f_k with a same variable ordering, and then apply factorization all together. A divisor extracted from f_k is used not only for f_k but also for all other outputs. In this way, all existing common divisors are factored, and all simple disjunctive decompositions are exploited.

One different point is that the fan-out free gates in single output decomposition may have a new fan-out between the different output functions, and then need ad-

Table 1: Experimental results.

circuit			SDD-opt				Sawada
name	in	out	lit. (ISOP)	lit. (final)	SDD	time† (sec)	time‡ (sec)
apex1(all)	45	45	3646	2434	1	59.0	-
apex1(po23)	39	1	550	293	2	1.8	>1000
apex2(all)	39	3	3439	272	5	5.9	-
apex2(po1)	36	1	1496	114	6	0.8	>1000
apex3(all)	54	50	3898	2212	0	44.3	-
apex3(po8)	45	1	506	334	3	2.1	>1000
apex6(all)	135	99	3928	1031	1	13.1	-
apex6(po3)	24	1	187	55	4	0.4	426.44
apex7(all)	49	37	1296	265	0	1.7	-
apex7(po4)	24	1	219	48	5	0.2	1.31
c432(all)	36	7	778064	1513	0	415.4	-
c432(po1)	18	1	18	18	10	0.0	-
c880(po24)	45	1	789137	×	×	×	>1000
dalu(po2)	47	1	1930	136	5	0.8	34.26
frg2(all)	143	139	25369	1030	1	19.2	-
frg2(po135)	25	1	433	47	10	0.9	0.80
seq(all)	41	35	12682	2498	1	67.8	-
seq(po6)	38	1	1400	279	4	2.2	>1000
too_large(po1)	36	1	1496	115	6	1.0	>1000
vda(po15)	17	1	183	103	0	0.5	20.26

†: SUN Ultra 30, ‡: SUN Ultra 1, cited from [12].
×: Memory out. (> 4M BDD nodes)

ditional factorization. For example, each single output function $f_1 = x_1x_2x_3 + y_1$ and $f_2 = x_1x_2\overline{x_3} + y_2$ do not need factorization, but if they are regarded as a multi-output function, the common literal set x_1x_2 should be factored. We added the procedure to check the common literal sets between multi-output functions.

4 Experimental Results

Based on the above method, we implemented a multi-level logic optimizer "SDD-opt," which extracts all simple disjunctive decompositions. BDD and ZBDD-based symbolic manipulation techniques[8] support fast execution for large-scale functions.

Table 1 shows experimental results for the circuits chosen from MCNC'91 benchmark. In this table, the column "lit.(ISOP)" shows the number of literals included in the irredundant SOP, generated by Minato-Morreale algorithm. "lit.(final)" is the total number of literals in the factored SOP network. The column "SDD" shows the number of simple disjunctive decompositions which are found in the final network. If the circuit has multiple primary outputs, we count the common decompositions for the multiple-output function. We compared our results with the restricted exhaustive search method by Sawada et al.[12], which also reported the number of all simple disjunctive decompositions[2].

The experimental results show that SDD-opt can extract and enumerate all simple disjunctive decompositions of the large-scale functions. We can see that our approach is practicable and efficient for extracting simple disjunctive decompositions.

Figure 6 shows the decomposition results for some selected examples. These lists indicate the partitions of primary inputs corresponding to the simple disjunctive decompositions. The bracket `[ ]` denotes an ordinary decomposition, and the parenthesis `( )` denotes a symmetric group of decompositions. For example, `(a b c)` implies `[[a b] c]`, `[[a c] b]`, and `[[b c] a]`. SDD-opt displays such a structural information after factorization. It is a useful information for analyzing functionality of the circuit, and can be utilized for technology mapping and circuit partitioning.

[2]LODE[5] is the latest previous work, but here we could not compare directly because the experiment in [5] enumerates the number of decomposable primary outputs, not the number of all decompositions.

```
apex6(po3): x74 [x28 x29 x38 x73 x75 x76 x80 x81
x92 x93 x84 x110 (x90 x91 [x33 x34 x35 x36 x87
x88 x89]) [x85 x86]]

apex7(po4): x28 [x16 [x31 x32 x42 [x2 x3 x4 x5
x6 x7 x10 x11 x12] (x39 x40 x41) (x43 x44 x45
x46 x47 x48 x49)]]

frg2(po135): (x41 x43 x58 [x139 (x140 x143 [x59
x60 x61 x62 x63 [x44 x49] [x45 x50] [x46 x51]
[x47 x52] [x48 x53]] (x134 x135 x136 x137))])

seq(po6): (x8 x33 x34 [x1 x2 x3 x4 x5 x6 x10
x12 x13 x14 x15 x16 x17 x18 x19 x20 x22 x23 x24
x25 x29 x30 x31 x32 x35 x36 x37 x38 x39 x40 x41
[x11 x28] [x26 x27]])

too_large(po1): x28 [x1 x2 x3 x8 x9 x10 x11 x13
x14 x15 x17 x20 x21 x22 x23 x24 x25 x26 x29 x30
x31 x32 x33 x34 x35 x36 [[x4 x18] [x12 x19]] (x5
x6 x7) [x16 x27]]
```

Figure 6: Decomposition results.

Table 2: Comparisons with other optimizers.

circuit			original	final literals		
name	in	out	literals†	LODE†	SIS†	SDDopt
9symml	9	1	277	76	223	104
CM150	21	1	77	47	51	62
PARITY	16	1	60	60	60	60
alu2	10	6	453	354	357	557
cmb	16	4	62	36	51	33
f51m	8	8	169	98	91	185
lal	26	19	221	134	105	112
mux	21	1	92	47	51	62
term1	34	10	624	165	197	147
ttt2	24	21	341	258	216	223
s1494	14	25	1393	793	661	816
s298	17	20	244	146	114	167
s526	24	27	445	257	191	238
s832	23	24	769	431	352	400

†: Cited from [5].

Not only finding simple disjunctive decompositions, SDD-opt gives a compact multi-level logic network for a given function. We compared the number of literals in final SOP networks with other optimizers LODE[5] and SIS[3]. LODE also exploits simple disjunctive decompositions based on BDD manipulation. Table 2 shows that our results are sometimes better and sometimes worse. SDD-opt guarantees to extract all simple disjunctive decompositions, but if the function is dominated by other non-simple or non-disjunctive decompositions, further optimization might be possible. Anyway, our results show that SOP-based factorization is strong enough to make a "first version" of compact network before applying more intensive optimization.

Lastly, we show an example where our method is very effective. As shown in Fig. 7, an n-bit adder has a simple disjunctive decomposition on each digit. SDD-opt can extract those decomposition points independently of the initial network structure, and thus a nearly minimum

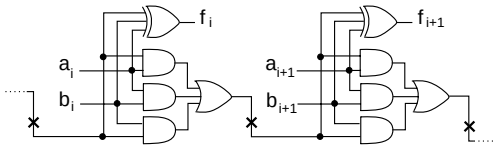


Figure 7: Decomposition of an n-bit adder.

design can be obtained.

5 Conclusion

In this paper, we presented a method to extract all simple disjunctive decompositions by generating irredundant SOPs and applying factorization. The experimental results show that we can quickly generate the multilevel SOP networks including all possible simple disjunctive decompositions.

Minato-Morreale algorithm has an important feature that the simple disjunctive decompositions are uniquely factored in the result of SOPs. Other SOP minimization algorithms, such as *ESPRESSO*[2], does not guarantee this uniformity.

Through out the discussion, we clarified the relationship between SOP-based factorization and functional decomposition. We can conclude that two-level logic minimization and factorization can be strong enough to perform simple disjunctive decomposition. In other words, the result of factorization will not be remarkably improved by any other strong optimization methods when the function is dominated by simple disjunctive decompositions. Thus, our result shows a guideline to the choice of logic optimization strategies.

As future work, it will be important to explore the techniques of non-simple or non-disjunctive decomposition.

References

[1] R. Ashenhurst, "The Decomposition of Switching Functions," *In Proc. an International Symposium on the Theory of Switching*, pp. 74-116, Apr. 1957.

[2] R. Brayton, G. Hachtel, C. McMullen, and A. S.-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis," Kluwer Academic Publishers, 1984.

[3] R. Brayton, R. Rudell, A. S.-Vincentelli, and A. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. on CAD*, 6(6), pp. 1062-1081, Nov. 1987.

[4] R.Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, C-35(8), pp. 667-691, Aug. 1986

[5] V.Bertacco, and M.Damiani, "The disjunctive Decomposition of Logic Functions," *In Proc ICCAD'97*, pp.78-82, Nov. 1997.

[6] Y. Lai, M. Pedram, and S. Vrudhula, "BDD Based Decomposition of Logic Functions with Application to FPGA Synthesis," *In Proc. DAC'93*, pp. 642-647, June 1993.

[7] S. Minato, "Fast Generation of Irredundant Sum-of-Products Forms from Binary Decision Diagrams," *In Proc. Synthesis and simulation Meeting and International Interchange (SASIMI'92)*, pp. 64-73, Apr. 1992.

[8] S. Minato, "Fast Factorization Method for Implicit Cube Set Representation", *IEEE Trans. on CAD*, 15(4), pp.377-384, Apr. 1996.

[9] E. Morreale, "Recursive Operators for Prime Implicant and Irredundant Normal Form Determination," *IEEE Trans. Comput.*, C-19(6), pp. 504-509, June 1970.

[10] J. Roth and R. Karp, "Minimization Over Boolean Graphs," *IBM Journal, pp. 227-238*, Apr. 1962.

[11] T. Sasao, "FPGA Design by Generalized Functional Decomposition," In T. Sasao, editor, *Logic Synthesis and Optimization*, pp. 233-258, Kluwer Academic Publishers, 1993.

[12] H. Sawada, S. Yamashita, and A. Nagoya, "Restricted Simple Disjunctive Decompositions Based on Grouping Symmetric Variables," *In Proc. Seventh Great Lakes Symposium on VLSI (GLSVLSI'97)*, pp. 39-44, Mar. 1997.

Appendix

(Minato-Morreale Algorithm)

GetISOP($f(X)$): /* $f(X) : \{0,1\}^n \rightarrow \{0,1,d\}$*/
  if ($f(X)$ always returns 0 or d) $isop \leftarrow 0$
  else if ($f(X)$ always returns 1 or d) $isop \leftarrow 1$
  else {
    $v \leftarrow$ one of variable in X
      /* choose top variable in BDD. */
    $f_1(X') \leftarrow f(X|_{v=1})$ /* $X' = X - \{v\}$. */
    $f_0(X') \leftarrow f(X|_{v=0})$ /* f_1, f_0 are cofactors of f. */
    $f_P(X') \leftarrow \begin{cases} d & \text{if } (f_1(X') = 1) \wedge (f_0(X') \neq 0) \\ f_1(X') & \text{otherwise} \end{cases}$
      /* f_P must be covered by cubes with v. */
    $isop_P \leftarrow$ GetISOP($f_P(X')$)
      /* generates $isop_P$ recursively. */
    $f_N(X') \leftarrow \begin{cases} d & \text{if } (f_0(X') = 1) \wedge (f_1(X') \neq 0) \\ f_0(X') & \text{otherwise} \end{cases}$
      /* f_N must be covered by cubes with $\overline{v}$. */
    $isop_N \leftarrow$ GetISOP($f_N(X')$)
      /* generates $isop_N$ recursively. */
    $f_1'(X') \leftarrow \begin{cases} d & \text{if already covered by } isop_P \\ f_1(X') & \text{otherwise} \end{cases}$
    $f_0'(X') \leftarrow \begin{cases} d & \text{if already covered by } isop_N \\ f_0(X') & \text{otherwise} \end{cases}$
    $f_D(X') \leftarrow \begin{cases} 0 & \text{if } (f_1'(X') = 0) \vee (f_0'(X') = 0) \\ d & \text{if } (f_1'(X') = d) \wedge (f_0'(X') = d) \\ 1 & \text{otherwise} \end{cases}$
      /* f_D must be covered by cubes without $v, \overline{v}$. */
    $isop_D \leftarrow$ GetISOP($f_D(X')$)
      /* generates $isop_D$ recursively. */
    $isop \leftarrow v \cdot isop_P + \overline{v} \cdot isop_N + isop_D$
  }
  return $isop$

(Proof of Theorem 3.2) The function f can be represented with a decomposition chart, as shown in Fig. 8. Since f is simple disjunctive decomposable, each column $c_1, c_2, \ldots, c_n (n = 2^{|Y|})$ of this chart must fall into one of the four functions: $h(X), \overline{h}(X), 0$, and 1.

We assume the variable ordering $x_1, x_2, \ldots, x_{|X|}$ for X, and $y_1, y_2, \ldots, y_{|Y|}$ for Y. In general, the two orderings are interleaved, for example, $(x_1, x_2, y_1, x_3, y_2, y_3, x_4, \ldots)$. Thus we should consider all cases related to the order of the variables.

1. In case of the variable ordering as $(x_1, \ldots, x_{|X|}, y_1, \ldots, y_{|Y|})$:

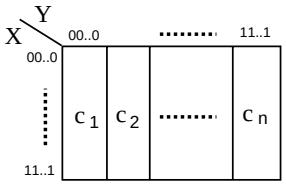


Figure 8: Decomposition chart of f.

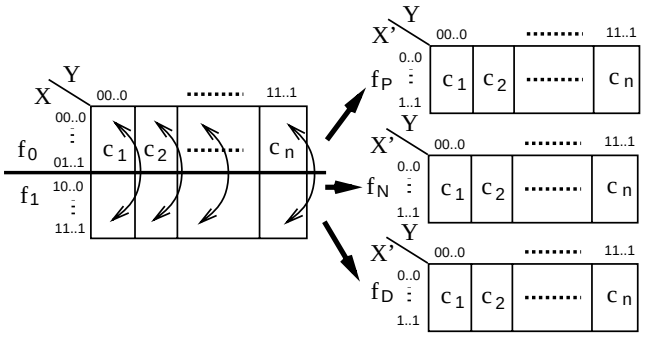


Figure 9: Expansion of f by x_i.

When we expand f by a variable in X, each column c_k of the decomposition chart is expanded independently, and three sub-functions are generated as shown in Fig. 9. After expanding by all the variables in X, each column yields one of the SOPs $isop^h(X)$, $isop^{\overline{h}}(X)$, 0, or 1. We then expand it by the variables in Y. Some of the variables in Y are appended to the SOPs and finally they combined into one SOP. Thus, the decomposition function $h(X)$ and its complement are represented uniquely as in Equation 2.

2. Next, we consider an interleaved ordering as $(x_1, \ldots, x_i, \quad y_j, \quad x_{i+1}, \ldots, x_{|X|}, \quad y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_{|Y|})$:

   When we expand f by a variables in $\{x_1, \ldots, x_i\}$, each column c_k of the decomposition chart is expanded independently, and three sub-functions are generated as shown in Fig. 9. After expanding by all the variables in $\{x_1, \ldots, x_i\}$ each column of sub-function contains one of functions $h^*(X)$, $\overline{h^*(X)}$, 0 or 1, where $h^*(X)$ and $\overline{h^*(X)}$ are the sub-function of $h(X)$, $\overline{h(X)}$, obtained by the previous expansions.

   Next, we expand it by y_j. As shown in the description of Minato-Morreale algorithm, we divide the function f into two cofactors f_1 and f_0, by assigning $y_j = 1, 0$. In this process, the decomposition chart is divided into two parts, as shown in Fig. 10. For computing sub-function f_P, f_N, and f_D, each corresponding columns in f_1 and f_0 are processed independently. We consider all possible combinations of f_0 and f_1 from the four functions $h^*(X)$, $\overline{h^*(X)}$, 0 and 1. In Fig. 11, we list all the combinations and their results (except symmetric cases).

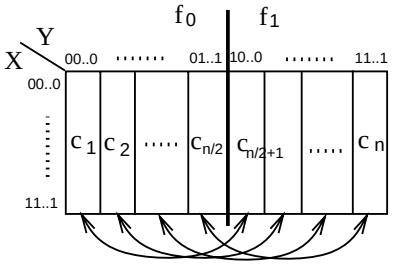


Figure 10: Expansion of f by y_j.

f_1	f_0	f_P	f_N	f_D
h^*	0	h^*	0	0
h^*	1	0	1	h^*
h^*	h^*	0	0	h^*
h^*	$\overline{h^*}$	h^*	$\overline{h^*}$	0
0	0	0	0	0
1	0	1	0	0
1	1	0	0	1

Figure 11: Combination of f_1, f_0 and f_P, f_N, f_D.

For all possible combinations of f_1 and f_0, no new sub-functions are produced in f_P, f_N, and f_D. After the expansion by y_j, each column containing $h^*(X)$ or $\overline{h^*(X)}$ are followed by the expansions by $\{x_{i+i}, \ldots, x_{|X|}\}$. Thus, interleaving y_j in X does not disturb the generation of $isop^h(X)$ and $isop^{\overline{h}}(X)$. The uniformity of decomposition function $h(X)$ and its complement are kept also for this variable ordering.

By considering different interleaving of variables, we can produce any variable ordering. Thus, Minato-Morreale algorithm guarantees the uniformity of decomposition function $h(X)$ and its complement, for any fixed variable ordering. □