

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL  
INSTITUTO DE INFORMÁTICA  
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VINICIUS CALLEGARO

**On the optimal minimization of special classes of Boolean functions**

Dissertation presented in partial fulfillment  
of the requirements for the degree of  
Doctor of Computer Science

Prof. Dr. André Inácio Reis  
Advisor

Porto Alegre, July 2016

## CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Callegaro, Vinicius

On the optimal minimization of special classes of Boolean functions / Vinicius Callegaro. – 2016.

130 f.:il.

Orientador: André Inácio Reis.

Tese (Doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2016.

1. Introduction. 2. Background 3. Classes of Boolean functions. 4. Read-Once Functions. 5. Disjoint-Support Decomposable Functions. 6. Read-Polarity-Once functions. 7. Conclusions. I. Reis, André Inácio. II. Ribas, Renato Perez. On the optimal minimization of special classes of Boolean functions.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento

Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb

Coordenador do PPGC: Prof. Luigi Carro

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

## ACKNOWLEDGEMENTS

I would like to dedicate this dissertation to my wife Monica, my parents Aldo and Marta, and to my sister Thalita who were essential to me during the doctoral years.

I am grateful to my advisor Professor Andre Inácio Reis and my co-advisor Professor Renato Perez Ribas for their knowledge and effort to make this dissertation possible. I also would like to thank Professor Marek Perkowski for his valuable suggestions and for advising me during my internship at Portland State University, Portland OR – US.

Special thanks go to my colleagues in the Logic Circuit Synthesis (LogiCS) lab who helped me with discussions, ideas, and fun.

I would like to thank the members of my thesis committee: Professors Marek Perkowski, Sérgio Bampi, and Marcelo de Oliveira Johann for their constructive comments which helped to improve this work.

The research presented in the dissertation was supported by Brazilian funding agencies CAPES, CNPq, and FAPERGS, under grant 11/2053-9 (Pronem).

## ABSTRACT

The problem of factoring and decomposing Boolean functions is  $\Sigma_2^P$ -complete for general functions. Efficient and exact algorithms can be created for an existing class of functions known as read-once, disjoint-support decomposable and read-polarity-once functions.

A factored form is called *read-once* (RO) if each variable appears only once. A Boolean function is RO if it can be represented by an RO form. For example, the function represented by  $F = x_1x_2 + x_1x_3x_4 + x_1x_3x_5$  is a RO function, since it can be factored into  $F = x_1(x_2 + x_3(x_4 + x_5))$ .

A Boolean function  $f(X)$  can be decomposed using simpler subfunctions  $g$  and  $h$ , such that  $f(X) = h(g(X_1), X_2)$  being  $X_1, X_2 \neq \emptyset$ , and  $X_1 \cup X_2 = X$ . A *disjoint-support decomposition* (DSD) is a special case of functional decomposition, where the input sets  $X_1$  and  $X_2$  do not share any element, *i.e.*,  $X_1 \cap X_2 = \emptyset$ . Roughly speaking, DSD functions can be represented by a read-once expression where the exclusive-or operator ( $\oplus$ ) can also be used as base operation. For example,  $F = x_1(x_2 \oplus (x_4 + x_5))$ .

A *read-polarity-once* (RPO) form is a factored form where each polarity (positive or negative) of a variable appears at most once. A Boolean function is RPO if it can be represented by an RPO factored form. For example the function  $F = \overline{x_1}x_2x_4 + x_1x_3 + x_2x_3$  is RPO, since it can be factored into  $F = (\overline{x_1}x_4 + x_3)(x_1 + x_2)$ .

This dissertation presents four new algorithms for synthesis of Boolean functions. The first contribution is a synthesis method for read-once functions based on a divide-and-conquer strategy. The second and third contributions are two algorithms for synthesis of DSD functions: a top-down approach that checks if there is an OR, AND or XOR decomposition based on sum-of-products, product-of-sums and exclusive-sum-of-products inputs, respectively; and a method that runs in a bottom-up fashion and is based on Boolean difference and cofactor analysis. The last contribution is a new method to synthesize RPO functions which is based on the analysis of positive and negative transition sets. Results show the efficacy and efficiency of the four proposed methods.

**Keywords:** Logic Synthesis, Factoring, Decomposition, Read-Once, Disjoint-Support Decomposition, Read-Polarity-Once.

# Minimização ótima de classes especiais de funções Booleanas

## RESUMO

O problema de fatorar e decompor funções Booleanas é  $\Sigma_2^P$ -completo para funções gerais. Algoritmos eficientes e exatos podem ser criados para classes de funções existentes como funções *read-once*, *disjoint-support decomposable* e *read-polarity-once*.

Uma forma fatorada é chamada de *read-once* (RO) se cada variável aparece uma única vez. Uma função Booleana é RO se existe uma forma fatorada RO que a representa. Por exemplo, a função representada por  $F = x_1x_2 + x_1x_3x_4 + x_1x_3x_5$  é uma função RO, pois pode ser fatorada em  $F = x_1(x_2 + x_3(x_4 + x_5))$ .

Uma função Booleana  $f(X)$  pode ser decomposta usando funções mais simples  $g$  e  $h$  de forma que  $f(X) = h(g(X_1), X_2)$  sendo  $X_1, X_2 \neq \emptyset$ , e  $X_1 \cup X_2 = X$ . Uma decomposição disjunta de suporte (*disjoint-support decomposition* – DSD) é um caso especial de decomposição funcional, onde o conjunto de entradas  $X_1$  e  $X_2$  não compartilham elementos, *i.e.*,  $X_1 \cap X_2 = \emptyset$ . Por exemplo, a função  $F = x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3\bar{x}_4 + x_1\bar{x}_2x_4$  é DSD, pois existe uma decomposição tal que  $F = x_1(x_2 \oplus (x_3 + x_4))$ .

Uma forma *read-polarity-once* (RPO) é uma forma fatorada onde cada polaridade (positiva ou negativa) de uma variável aparece no máximo uma vez. Uma função Booleana é RPO se existe uma forma fatorada RPO que a representa. Por exemplo, a função  $F = \bar{x}_1x_2x_4 + x_1x_3 + x_2x_3$  é RPO, pois pode ser fatorada em  $F = (\bar{x}_1x_4 + x_3)(x_1 + x_2)$ .

Esta tese apresenta quatro novos algoritmos para síntese de funções Booleanas. A primeira contribuição é um método de síntese para funções *read-once* baseado em uma estratégia de divisão-e-conquista. A segunda contribuição é um algoritmo *top-down* para síntese de funções DSD baseado em soma-de-produtos, produto-de-somas e soma-exclusiva-de-produtos. A terceira contribuição é um método *bottom-up* para síntese de funções DSD baseado em diferença Booleana e cofatores. A última contribuição é um novo método para síntese de funções RPO que é baseado na análise de transições positivas e negativas.

**Palavras-chave:** Síntese Lógica, Fatoração, Decomposição, *Read-Once*, *Disjoint-Support Decomposition*, *Read-Polarity-Once*.

## FIGURES

Figure 1 – Example of an arbitrary Boolean function using truth table (left) and Karnaugh map (right) representations. The same Boolean function can be represented by the bit stream $f(x_1, x_2, x_3) = 11001001_2$ .	7
Figure 2 – Binary decision diagram (BDD) representation of a Boolean function.	8
Figure 3 – A reduced and ordered binary decision diagram (ROBDD) of a Boolean function.	8
Figure 4 – Example of And-Inverter-Graph (AIG). Complemented edges are denoted by small dots. Notice that primary input nodes $\{x_1, x_2, x_3\}$ have no incoming edges. Although being structurally distinct, both AIGs represent the same function.	9
Figure 5 – Karnaugh-map and covering example for $F = x_1x_2 + x_2x_3 + \overline{x_1x_2x_3}$	15
Figure 6 – Karnaugh-map and a minimum covering example for function $F_2 = \overline{x_1x_2x_4} + \overline{x_2x_3x_4} + \overline{x_1x_2x_3} + x_1x_3\overline{x_4}$	16
Figure 7 – ESOP covering resulting $F = x_2 \oplus \overline{x_1} \cdot \overline{x_3}$	17
Figure 8 – Examples of switch networks: a) a switch controlled by the variable $x_1$ b) a series association representing $x_1 \cdot x_2$ c) a parallel association representing $x_1 + x_2$ d) a series-parallel network, created by series and parallel associations representing $x_1 \cdot (x_2 + x_3)$ and e) a non-series-parallel network representing $(x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_5 + x_4 \cdot x_3 \cdot x_2 + x_4 \cdot x_5)$ .	19
Figure 9 – A rooted tree (a) and its respective series-parallel switch network (b) representing the RO function $F = x_1x_2 + x_3x_4 + x_5$ .	23
Figure 10 – Rooted tree (a) and its respective series-parallel switch (b) network representing the RPO function $F = x_1x_4 + x_3x_1 + x_2$ .	24
Figure 11 – Schematic of (a) disjoint and (b) non-disjoint decompositions. SOURCE: (SASAO; BUTLER, 1997).	25
Figure 12 – Schematic examples of (a) disjoint, (b) strong and (c) weak bi-decompositions. SOURCE: (MISHCHENKO; STEINBACH; PERKOWSKI, 2001).	26
Figure 13 – Comparison of unate, read-once (RO), full disjoint-support bi-decomposition (DSD) and read-polarity-once (RPO) classes. As can be seen, the class of RO functions is a subset of both DSD and RPO classes.	27
Figure 14 – Example of a non-collapsed tree (left) and the resulting tree after the collapse operation (right).	31
Figure 15 – Example of sorting trees. Original tree (left) and sorted tree with input order $Y = [x_1, x_2, x_3, x_4, x_5, x_6]$ (right).	31
Figure 16 – Example explicitly showing the hidden part from 13-20 (Table 7). The top part (cloud) of the tree is hidden without lack of generality and just for presentation sake.	33

Figure 17 – Two examples of the <b>RO_BY_COFACTOR</b> algorithm using the input function $F = x_1x_2 + x_1x_3$ . A full BDD (left) and an ROBDD (right) representing $f$ are shown. ....	41
Figure 18 – Lowest common ancestor (LCA) example: The complete LCA for the variable set $x_1, x_2, x_4$ is node $n_2$ . Node $n_4$ is the LCA for $x_4, x_5, x_6$ . For this example, there is not a complete LCA for $x_4, x_5, x_6$ . ....	42
Figure 19 – Step by step example of removing variables $\Delta = \{x_2, x_3, x_6, x_7\}$ from a read-once tree $T = x_2 + x_3 + x_4x_5 + x_6x_7$ . (a) Original tree. (b) Node $x_2$ removed. (c) Node $x_3$ removed. (d) Node $x_6$ removed. (e) Node $x_7$ removed. (f) Operator node “.” simplified. (g) Operator node “+” simplified. ....	47
Figure 20 – Runtime analysis of the <b>RO_COMPOSITION</b> method. The number of inputs are in thousands. Results support the claim that the <b>RO_COMPOSITION</b> runs in $O(n)$ . ....	53
Figure 21 – ROBDD representing a non-read-once function $F_1 = x_1x_3x_4x_5x_6 + x_2x_3x_4x_5x_6 + x_1x_2x_5x_6 + x_2x_3x_5x_6 + x_1x_3x_5x_6 + x_1x_2x_3x_4x_5 + x_1x_2x_4x_6 + x_1x_2x_3x_4x_5 + x_1x_3x_5x_6 + x_1x_2x_3x_5 + x_2x_3x_4x_5x_6 + x_1x_2x_3x_4 + x_1x_2x_4x_5x_6 + x_1x_2x_4x_5x_6 + x_1x_3x_4x_5 + x_1x_2x_3x_4x_6$ . Nodes in green represent read-once functions. ....	54
Figure 22 – A variable intersection graph obtained from an (a) ISOP $F = (!a \cdot !b \cdot c \cdot d + a \cdot !d + b \cdot !c + a \cdot !c + b \cdot !d)$ and (b) from ESOP form $H = 1 \oplus (!a \cdot !b) \oplus (c \cdot d)$ . ....	59
Figure 23 – A generic DSD tree (a) and its corresponding disconnected VIG (b). ....	61
Figure 24 – A complete execution tree of the proposed algorithm. ....	64
Figure 25 – Illustration of $F(X) = H(G(X_1), X_2)$ . Functions $G$ and $H$ are called composition and decomposition functions of $F$ , respectively. The variable set $X_1$ and $X_2$ are named bound and free sets, respectively. ....	70
Figure 26 – Two examples of partial-DSD functions. A top-down approach can find the XOR decomposition on top of (a), while a bottom-up approach could not. In (b), a bottom-up approach identifies the AND, OR and XOR compositions, while a top-down approach finds nothing. ....	70
Figure 27 – Two examples of full-DSD functions when considering AND, XOR and MUX as basis. The example shown in (a) is the implementation of the output 04 of the <i>shift</i> circuit, taken from the (ESPRESSO BOOK EXAMPLES). The circuit depicted in (b) is the implementation of the <i>mux</i> circuit, present in the ACM/SIGDA (MCNC) benchmark (IWLS, 2005). ....	71
Figure 28 – A full-tree model, representing the worst-case of a full-DSD function. ....	80
Figure 29 – Comparison of the state-of-the-art algorithms available in ABC tool on a full-DSD benchmark. ....	82
Figure 30 – Example a read-polarity-once function $f = (!a \cdot d + c) \cdot (a + b)$ . ....	86
Figure 31 – Example of an RPO function. ....	88

Figure 32 – Step by step example of the proposed factoring algorithm for RPO functions. The initial graph (a) contains the relationship between literals, where solid edges represent AND grouping while dashed edges represent OR grouping. The algorithm proceeds and chooses the solid edge between “!a” and “d”, leading to the graph shown in (b). The algorithm continues with (c) and (d) steps, reaching the optimal solution in (e). .....	90
Figura 1 – Análise de tempo de execução do método <b>RO_COMPOSITION</b> . Os resultados mostram que o método roda em $O(n)$ . .....	114
Figura 2 – ROBDD representando uma função não- <i>read-once</i> $F1 = x1x3x4x5x6 + x2x3x4x5x6 + x1x2x5x6 + x2x3x5x6 + x1x3x5x6 + x1x2x3x4x5 + x1x2x4x6 + x1x2x3x4x5 + x1x3x5x6 + x1x2x3x5 + x2x3x4x5x6 + x1x2x3x4 + x1x2x4x5x6 + x1x2x4x5x6 + x1x3x4x5 + x1x2x3x4x6$ . Nodos em verde representam funções <i>read-once</i> .....	115
Figura 3 – Um grafo de interseção de variáveis (VIG) obtido de uma (a) ISOP $F = (!a \cdot !b \cdot c \cdot d + a \cdot !d + b \cdot !c + a \cdot !c + b \cdot !d)$ e (b) de uma forma ESOP $H = 1 \oplus (!a \cdot !b) \oplus (c \cdot d)$ . .....	117
Figura 4 – Uma árvore de execução completa do algoritmo proposto. ....	119
Figura 5 - Comparação entre o método proposto e os métodos estado-da-arte para decomposição DSD disponíveis na ferramenta ABC. O tempo de execução do método proposto é representado por triângulos azuis. ....	124



## TABLES

Table 1 – Fundamental operations over sets. The gray area represents the result of the operation. ....	5
Table 2– Truth table representing the complement operation. ....	10
Table 3 – Truth table representing sum, product and exclusive-sum operations over two Boolean functions $f$ and $g$ . ....	10
Table 4 – Truth table representing every negative and positive cofactor for each input variable. The last column shows the cube-cofactor of $f$ w.r.t the input variables $x_1$ and $x_2$ . ....	11
Table 5 – Example of minterms and maxterms enumeration of a function $f(X)$ (same presented in Figure 1) depending of three input variable $X = \{x_1, x_2, x_3\}$ . ....	14
Table 6 – Enumeration of classes of functions: read-once (RO), full disjoint-support bi-decomposition (DSD) and read-polarity-once (RPO). ....	28
Table 7 – Enumeration of all possible read-once trees regarding variable $x_i$ 's position. For each case, a read-once tree is depicted in column $F$ , and negative and positive cofactors w.r.t input $x_i$ are shown in column $F_{xi} = 0$ and $F_{xi} = 1$ , respectively. Nodes depicted by triangles denote read-once subtrees. Cases 1-13 represent the complete read-once tree. In cases 13-20, part of the read-once tree is hidden, without lack of generalization. The table shows 20 cases of interest. ....	34
Table 8 – Enumeration of all possible read-once trees considering the complete lower common ancestor node nCLCA. Nodes depicted by triangles denote read-once subtrees. Filled triangles represent those subtrees $T_i$ where all support variables are missing, i.e. $\text{sup}(T_i) \subseteq \Delta$ . ....	50
Table 9 – Runtime results after running <b>RO_BY_COFACTOR</b> over ROBDDs representing read-once functions. The ROBDDs were constructed using a random variable order. ....	55
Table 10 – Results for a benchmark composed of all DSD functions up to 6 inputs, grouped by equivalence through input permutation (P-classes). ....	65
Table 11 – Results of decompositions over ESPRESSO book PLA benchmark. ....	66
Table 12 – Comparison of DSD methods considering two sets of full-DSD functions. ....	81
Table 13 – Comparison between different factoring approaches. ....	85
Table 14 – Summary of components used in the proposed decomposition. The negative (positive) cofactor w.r.t $x_i$ is represented by $f_{xi} = 0$ ( $f_{xi} = 1$ ). ....	92
Table 15 – Total number of literals and runtime obtained when factoring 1,462 RPO functions using different approaches. ....	94
Table 16 – Decomposition of circuits into $k$ -cuts, with $k=6$ , $k=8$ and $k=10$ . The number of read-once (RO), disjoint support decomposition (DSD) and read-polarity-once (RPO) functions is presented. ...	96
Table 17 – Results on the analysis of non-RPO functions over a set of benchmark functions. Column RPO show the number of RPO functions. The number of functions that have 1-dist Shannon, Davio or	

Quantifier-Based expansions are shown in columns Shannon, Davio and Quantifier-Based, respectively. Column 1-dist RPO shows the number of functions that have at least one of the previous decompositions. Finally, column RPO + 1-dist RPO represent the number of functions that are RPO or have 1-dist decomposition. ....97

Tabela 1 – Tempo de execução do método **RO\_BY\_COFACTOR** sobre ROBDDs representando funções *read-once*. Os ROBDDs foram construídos utilizando uma ordem randômica de variáveis. 116

Tabela 2 – Resultados para um conjunto de funções composto de funções DSD de até 6 variáveis, agrupado por equivalência através de permutação de entrada (classes-P). ....120

Tabela 3 – Circuitos selecionados do benchmark (ESPRESSO BOOK). ....121

Tabela 4 – Comparação de métodos de decomposição DSD considerando dois conjuntos de funções *full-DSD*. ....124

Tabela 5 – Enumeração de classes de funções Booleanas: *read-once* (RO), decomposição disjunta de suporte (DSD) e *read-polarity-once* (RPO). ....125

Tabela 6 – Número total de literais e tempo de execução para obter formas fatoradas para 1,462 funções RPO utilizando diferentes abordagens. ....126

Tabela 7 – Decomposição de circuitos em funções de até K entradas, com K variando de 6 a 10. O número de funções *read-once* (RO), decomposição disjunta de suporte (DSD) e *read-polarity-once* (RPO) é apresentado. ....128

Tabela 8 – Análise de funções RPO e não-RPO. As colunas Shannon, Davio e Quantifier-Based mostram o número de funções dos respectivos benchmarks que tem decomposição 1-dist RPO usando as expansões de Shannon, Davio e Quantifier-Based, respectivamente. A coluna 1-dist RPO mostra o número de funções que tem ao menos uma das decomposições anteriores. Finalmente, a coluna RPO + 1-dist RPO representa o número de funções que são RPO ou tem uma decomposição 1-dist RPO....129

## LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-Inverter-Graph
ASIC	Application-Specific Integrated Circuit
BDD	Binary Decision Diagram
CMOS	Complementary Metal Oxide Semiconductor
DSD	Disjoint-Support Decomposition
EDA	Electronic Design Automation
ESOP	Exclusive-Sum-Of-Products
FPGA	Field-Programmable Gate Array
GF	Good Factor
HDL	Hardware Description Language
IC	Integrated Circuit
LCA	Lowest Common Ancestor
LCI	Literal Cluster Intersection
MDC	Minimum Decision Chain
NPN	Negation-Permutation-Negation
NSP	Non-Series-Parallel
PLA	Programmable Logic Array
POS	Product-of-sums
QF	Quick Factor
RO	Read-Once
ROBDD	Reduced and Ordered Binary Decision Diagram
RPO	Read-Polarity-Once
RTL	Register Transfer Level
SOP	Sum-Of-Products
SP	Series-Parallel
VIG	Variable Intersection Graph
VLSI	Very-Large-Scale Integration

# SUMMARY

<b>ACKNOWLEDGEMENTS</b> .....	<b>11</b>
<b>ABSTRACT</b> .....	<b>12</b>
<b>RESUMO</b> .....	<b>13</b>
<b>FIGURES</b> .....	<b>14</b>
<b>1 INTRODUCTION</b> .....	<b>1</b>
1.1 Motivation and challenges.....	2
1.2 Objective.....	3
1.3 Text organization .....	4
<b>2 BACKGROUND</b> .....	<b>5</b>
2.1 Set theory.....	5
2.2 Boolean functions .....	6
2.2.1 Representing Boolean functions .....	6
2.2.2 Basic operations over Boolean functions.....	9
2.2.3 Unateness analysis .....	12
2.2.4 Shannon expansion.....	12
2.3 Boolean expressions.....	13
2.3.1 Canonical sum-of-products and product-of-sums .....	13
2.3.2 Two-level minimization .....	14
2.3.3 Factored forms .....	17
2.4 Switch networks .....	18
<b>3 CLASSES OF BOOLEAN FUNCTIONS</b> .....	<b>20</b>
3.1 Unate classes of functions.....	20
3.2 Symmetric Boolean functions.....	21

<b>3.3</b>	<b>Read-Once Boolean functions .....</b>	<b>21</b>
<b>3.4</b>	<b>Read-Polarity-Once Boolean functions.....</b>	<b>23</b>
<b>3.5</b>	<b>Boolean decomposition.....</b>	<b>24</b>
<b>3.6</b>	<b>Comparison of Boolean function classes.....</b>	<b>26</b>
<b>4</b>	<b>READ-ONCE FUNCTIONS .....</b>	<b>29</b>
<b>4.1</b>	<b>Previous work.....</b>	<b>29</b>
<b>4.2</b>	<b>Proposed method for Read-Once synthesis.....</b>	<b>30</b>
4.2.1	Notation and definitions .....	30
4.2.2	Read-once by cofactor composition .....	32
4.2.3	Composing read-once cofactor trees .....	42
<b>4.3</b>	<b>Results .....</b>	<b>53</b>
<b>4.4</b>	<b>Conclusions.....</b>	<b>56</b>
<b>5</b>	<b>DISJOINT-SUPPORT DECOMPOSABLE FUNCTIONS .....</b>	<b>57</b>
<b>5.1</b>	<b>Previous work.....</b>	<b>58</b>
<b>5.2</b>	<b>Top-down decomposition based on SOP, POS and ESOP forms .....</b>	<b>58</b>
5.2.1	Definitions and notation .....	59
5.2.2	Proposed method.....	60
5.2.3	A complete example of the proposed approach .....	63
5.2.4	Experimental results.....	64
5.2.5	Conclusions .....	67
<b>5.3</b>	<b>Bottom-up decomposition based on Boolean difference and cofactor analysis .....</b>	<b>67</b>
5.3.1	Definitions and notation .....	67
5.3.2	Bottom-Up Decomposition Properties .....	70
5.3.3	AND decomposition.....	72
5.3.4	XOR decomposition .....	72
5.3.5	MUX decomposition .....	73
5.3.6	Proposed full-DSD synthesis method.....	76
5.3.7	Experimental Results.....	80
5.3.8	Conclusions .....	82

<b>6</b>	<b>READ-POLARITY-ONCE FUNCTIONS</b> .....	<b>83</b>
6.1	Boolean function representation .....	84
6.2	Definition and properties of read-polarity-once functions .....	85
6.3	Proposed method for synthesis of RPO functions.....	86
6.3.1	Positive and negative transitions of a variable .....	87
6.3.2	Intuition for grouping variables by transition test .....	87
6.3.3	Literals and grouping definition .....	89
6.3.4	Literal cluster intersection graph.....	89
6.4	Decomposing non-RPO functions .....	90
6.4.1	Proposed decompositions .....	92
6.5	Experimental results .....	93
6.6	Conclusions.....	95
<b>7</b>	<b>CONCLUSIONS</b> .....	<b>98</b>
	<b>REFERENCES</b> .....	<b>100</b>

# 1 INTRODUCTION

The circuit synthesis design flow is usually divided into three major steps: architectural synthesis, logic synthesis and physical synthesis. Architectural synthesis, often called high-level synthesis, consists of transforming an algorithmic description of the desired behavior into a hardware format that implements that behavior, as in RTL (Register Transfer Level) format. Usually, those algorithmic descriptions are represented in a C-like format (e.g. System C) or a behavioral Hardware Description Language (HDL), e.g. VHDL or Verilog format.

The logic synthesis process has been one of the most commercially successful areas of electronic design automation (EDA). Most digital devices that we use in our day-to-day life have been designed by a set of logic synthesis tools. The logic synthesis task consists of several steps. These steps may differ according to the nature of the circuit, e.g. sequential or combinational. The main goal of logic synthesis is to determine the primitive structure of a circuit, i.e. its gate-level representation. It is typically divided into three phases: technology independent optimizations, technology mapping and technology dependent optimizations (MICHELI, 1994). The first one applies transformations that do not depend on the technology, but on the functional behavior of a Boolean network, e.g. factorization and decomposition algorithms. The technology mapping phase matches portions of the circuit to a cell with technology information. The technology dependent phase applies optimizations in the mapped circuit, such as cell resizing and logic duplication.

Physical synthesis, or geometrical level synthesis, consists mainly of two major tasks. Block placement physically distributes the cells. Wire routing connects the signals. (ALPERT; MEHTA; SAPATNEKAR, 2008).

This work addresses synthesis of Boolean functions in the scope of a digital circuit design flow, more precisely in the logic synthesis phase. It may also have broader scope since this work proposes algorithms for classes of Boolean functions that may have application in

different areas other than circuit synthesis, for example learning theory (ANGLUIN; HELLERSTEIN; KARPINSKI, 1993; BSHOUTY; HANCOCK; HELLERSTEIN, 1995) and databases (SEN; DESHPANDE; GETOOR, 2010) (KANAGAL; LI; DESHPANDE, 2011).

## 1.1 Motivation and challenges

The process of factoring Boolean functions is a fundamental operation in algorithmic logic synthesis (BRAYTON, 1987; HACHTEL; SOMENZI, 2006). Factoring is the process of deriving a parenthesized algebraic equation, or factored form, representing a given logic function. For instance,  $F = x_1x_2 + x_1x_3x_4 + x_1x_3x_5$  can be factored into the logically equivalent equation  $F = x_1(x_2 + x_3(x_4 + x_5))$ .

Any given logic function can be represented by distinct factored expressions. The task of factoring Boolean functions into shorter, more compact logically equivalent formulae is one of the basic operations at the early stages of algorithmic logic synthesis (HACHTEL; SOMENZI, 2006). In most design styles, like conventional CMOS gates, the device count for realizing a Boolean function corresponds almost directly to its factored equation in terms of literal count. Generating an optimum factored form, i.e. the shortest length equation, is a  $\Sigma_2^P$ -complete problem (GOLUMBIC; MINTZ, 1999). Hence, heuristic algorithms have been developed in order to obtain good factored solutions (BRAYTON, 1987; STANION; SECHEN, 1994; MINTZ; GOLUMBIC, 2005; HACHTEL; SOMENZI, 2006; YOSHIDA; FUJITA, 2011). Some well-known heuristic algorithms include X-Factor (MINTZ; GOLUMBIC, 2005), which provides good results but does not guarantee the minimal equations. In (LAWLER, 1964), the author claims to provide the exact factoring. However, Lawler's method is not scalable and becomes impractical even for some functions with only four inputs. Recently, new approaches have improved the factoring process for exact solutions, but the scalability and runtime still remain the main bottlenecks (YOSHIDA; IKEDA; ASADA, 2006; YOSHIDA; FUJITA, 2011; MARTINS ET AL., 2012).

Since optimal factoring and decomposition for general functions is a  $\Sigma_2^P$ -complete problem (BUCHFUEHRER; UMANS, 2011), a good strategy is to identify classes of Boolean functions that are easier to synthesize. This is the case of read-once, disjoint-support decomposition and read-polarity-once classes of functions.

A factored form is called *read-once* (RO) if each variable appears only once. A Boolean function is RO if it can be represented by an RO form (HAYES, 1975). For example, the



Boolean function represented by  $F = x_1x_2 + x_1x_3x_4 + x_1x_3x_5$  is a RO function, since it can be factored into  $F = x_1(x_2 + x_3(x_4 + x_5))$ .

A Boolean function  $f(X)$  can be decomposed using simpler subfunctions  $g$  and  $h$ , such that  $f(X) = h(g(X_1), X_2)$  being  $X_1, X_2 \neq \emptyset$ , and  $X_1 \cup X_2 = X$  (ASHENHURST, 1957), (CURTIS, 1962). A *disjoint-support decomposition* (DSD) is a special case of functional decomposition, where the input sets  $X_1$  and  $X_2$  do not share any element, *i.e.*,  $X_1 \cap X_2 = \emptyset$ . Roughly speaking, DSD functions can be represented by a read-once expression where the exclusive-or operator ( $\oplus$ ) can also be used as base operation. For example,  $F = x_1(x_2 \oplus (x_4 + x_5))$ .

A *read-polarity-once* (RPO) form is a factored form where each polarity (positive or negative) of a variable appears at most once. A Boolean function is RPO if it can be represented by an RPO factored form (CALLEGARO ET AL, 2012) For example the function  $F = \overline{x_1}x_2x_4 + x_1x_3 + x_2x_3$  is RPO, since it can be factored into  $F = (\overline{x_1}x_4 + x_3)(x_1 + x_2)$ .

The motivation of researching these special classes of functions is that, besides being simpler to synthesize, such classes are of special interest in the context of digital circuit design, since they are extremely frequent in circuit applications (PEER; PINTER, 1995), (MISHCHENKO, BRAYTON, 2013) (CALLEGARO ET AL, 2014). The challenge is, therefore, to create efficient and exact methods that can handle these function classes.

## 1.2 Objective

This dissertation introduces four new algorithms for synthesis of Boolean functions. The first contribution is a synthesis method that finds a read-once realization for a target function. The method was designed based on a divide-and-conquer strategy. Finding a read-once tree for a target function consists of obtaining read-once trees for simpler sub-problems: negative and positive cofactors (division phase). These solutions are then composed (conquer phase), resulting in a read-once solution for the original problem (target function). The method is independent of the Boolean function's data structure representation. It relies only on cofactor operation and equivalence checking regarding constants.

The second and third methods are algorithms for synthesis of DSD functions (CALLEGARO ET AL, 2015a). A top-down approach checks if there is an OR, AND, or XOR decomposition based on sum-of-products, product-of-sums and exclusive-sum-of-

products inputs, respectively. The another method runs in a bottom-up fashion and is based on Boolean difference and cofactor analysis (CALLEGARO ET AL, 2015b). Two simple tests provide sufficient and necessary conditions to identify AND and exclusive-OR (XOR) decompositions.

The last contribution is a new method to synthesize RPO functions (CALLEGARO ET AL, 2013). The method is based on the concept of positive and negative transition sets possible for each variable. The method is able to detect if two literals must be grouped through an AND or OR logic operation by examining transition sets.

### 1.3 Text organization

Chapter 2 – *Background* – Provides to the reader basic and consolidated knowledge for full understanding of the contributions presented in this work.

Chapter 3 – *Classes of Boolean functions* – Presents and overviews definitions and comparison of the main classes of functions that are discussed in this work.

Chapter 4 – *Read-Once Functions* – Presents a synthesis method that finds a read-once realization for a target function. The method was designed based on a divide-and-conquer strategy. Finding a read-once tree for a target function consists of obtaining read-once trees for simpler sub-problems: negative and positive cofactors. These solutions are then composed (conquer phase), resulting in a read-once solution for the original problem (target function). Results show the scalability of the proposed method.

Chapter 5 – *Disjoint-Support Decomposable Functions* – Two algorithms for synthesis of DSD functions are discussed. A top-down approach checks if there is an OR, AND, or XOR decomposition based on sum-of-products, product-of-sums and exclusive-sum-of-products inputs, respectively. The second method runs in a bottom-up fashion and is based on Boolean difference and cofactor analysis. Two simple tests provide sufficient and necessary conditions to identify AND and exclusive-OR (XOR) decompositions. Comparison with the state-of-the-art methods is performed, showing the efficiency of the proposed methods.

Chapter 6 – *Read-Polarity-Once functions* – A method based on the concept of positive and negative transition sets possible for each variable is presented. The method is able to detect if two literals must be grouped through an AND or OR logic operation by computing transition sets. Results of several experiments are also presented and discussed.

Chapter 7 – *Conclusions* – Summarizes the major contributions of this work.

## 2 BACKGROUND

This chapter introduces notation and preliminaries necessary to the understanding of this work. It gives to the reader a brief description of Boolean algebra and switching theory domain.

### 2.1 Set theory

This section presents basic concepts of set theory. This includes concepts of membership, sets, subsets and associated operations.

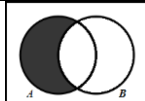
*Set:* a collection of distinct elements. The usual way of describing a set is by defining the characteristics of the elements belonging to it. For instance, if the set  $A$  is defined as the set of all positive even elements, the set  $A$  is completely defined. However, it is not possible to explicitly list all the elements in this set, as the number of elements is infinite. The set  $A$  could be described as  $A = \{2, 4, 6, 8, 10, \dots\}$ .

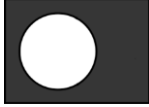
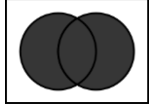
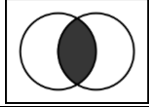

*Membership:* If an element  $a$  is member of the set  $A$ , write  $a \in A$ . For example, let  $B$  be a binary set such that  $B = \{0, 1\}$  which means that  $0 \in B$  and  $1 \in B$ .

*Subset:* Let  $A$  and  $B$  be two sets. We say  $B \subseteq A$  if all elements in  $B$  are also in the set  $A$ . In this case, we say that  $B$  is a subset of  $A$ , or equivalently  $A$  is a superset of  $B$ .

Table 1 presents some fundamental operations over sets and their respective Venn diagrams. The goal of this review is to present some very basic definitions and operations. For more information or formalism, (HALMOS, 1960) is suggested.

Table 1 – Fundamental operations over sets. The gray area represents the result of the operation.

Operation	Venn Diagram
<i>Set difference:</i> $A \setminus B$ results in the elements that are in $A$ but not in $B$ (in that order). Set difference is also known as $A$ butNot $B$ .	

Operation	Venn Diagram
<i>Complement:</i> If a universe $U$ is defined, $A^c$ results in a set that contains all elements of a universe $U$ that are not in $A$ : $U \setminus A$ .	
<i>Union:</i> $A \cup B$ results in a set that contains all elements that are member of either $A$ or $B$ .	
<i>Intersection:</i> $A \cap B$ results in a set that contains all elements that are members of both $A$ and $B$ .	
<i>Symmetric difference:</i> $A \Delta B$ results in a set that contains all elements which are in either of the sets but not in their intersection, or more formally $(A \cup B) \setminus (A \cap B)$ .	

## 2.2 Boolean functions

An  $n$ -input *Boolean function*  $f(X)$  defined by the variable (support) set  $X = \{x_0, \dots, x_{n-1}\}$  is a mapping

$$f(X): B^n \rightarrow B, \quad (1)$$

where  $B = \{0, 1\}$ . An element  $m \in B^n$ , i.e. an  $n$ -bit vector, is called *minterm*. There are  $2^n$  minterms in  $B^n$ . The *on-set* of function  $f$  comprises all minterms  $m$  such that  $f(m) = 1$  and is denoted  $\text{ON-SET}(f)$ . Conversely, the set representing all minterms such that  $f(m) = 0$  is called the *off-set* and is denoted  $\text{OFF-SET}(f)$ . Notice that a Boolean function can be uniquely represented by its on-set or off-set. A constant function  $\mathbf{1}$  has an empty off-set, while the constant function  $\mathbf{0}$  contains no element in the on-set. In this work, function and Boolean function are used interchangeably unless otherwise stated.

### 2.2.1 Representing Boolean functions

There are several ways to represent Boolean functions. One concern is regarding space complexity (memory consumption). Another concern is canonicity. A representation is said to be *canonical* if every function has a unique representation (BRYANT, 1986). Examples of canonical forms are truth-tables, ordered Karnaugh maps and Reduced and Ordered Binary Decision Diagrams (ROBDD). Among non-canonical data structures are Karnaugh maps (without ordering), Binary Decision Diagrams (BDD) (with no fixed order) and And-Inverter-Graphs (AIG).

### 2.2.1.1 Truth-table

The most straightforward way to representing functions is the *truth table*. In this representation, the output value of a function is specified for each possible input vector. For example, let the function  $f(x_1, x_2, x_3)$  be represented by the truth table shown in Figure 1 (left). The minterms  $\{000, 011, 110, 111\}$  are in the on-set of  $f$  while  $\{001, 010, 100, 101\}$  are in the off-set.

### 2.2.1.2 Karnaugh map

Another well-known approach of representing Boolean functions is the Karnaugh map (K-map) (KARNAUGH, 1953). The cells in the K-map are ordered using the *Gray code* (GRAY, 1953) such that the position of neighbor cells differs by exactly one bit. The function represented by Figure 1 (left) can be represented by the K-map depicted in Figure 1 (right).

It is also possible to represent the same function as a *bit string*, where the most significant bit (minterm 111) is on the left and the least significant bit (minterm 000) is on the right side, e.g.  $F(x_1, x_2, x_3) = 11001001_2$ .

$x_1$	$x_2$	$x_3$	$f$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

	$x_1x_2$				
$x_3$		00	01	11	10
0		1	0	1	0
1		0	1	1	0

Figure 1 – Example of an arbitrary Boolean function using truth table (left) and Karnaugh map (right) representations. The same Boolean function can be represented by the bit stream

$$f(x_1, x_2, x_3) = 11001001_2.$$

Although being a very simple way of representing Boolean functions, the truth table data structure is not scalable in practice, since it always uses  $2^n$  bits to store the information. For functions with more than 20 input variables, representing Boolean functions using a truth table data structure starts to be infeasible. In order to overcome this limitation, Akers proposed the concept of decision diagrams.

### 2.2.1.3 Binary decision diagrams

A *Binary Decision Diagram (BDD)* is a rooted, directed graph with vertex set  $V$  containing two types of vertices. A *nonterminal* vertex  $v$  has as attributes an input variable  $x_i$  and two children  $low(v)$ ,  $high(v) \in V$ . Each nonterminal vertex  $v$  behaves like an *if-then-else* operator, i.e. if  $x_i = \mathbf{1}$  then go to  $high(v)$  else go  $low(v)$ . A *terminal* vertex  $v$  has as attribute a value  $c \in \{0, 1\}$  (AKERS, 1978). The same function represented in Figure 1 could be represented by a BDD as depicted in Figure 2. The dashed edges represent the low child nodes, while the non-dashed lines are the high child nodes.

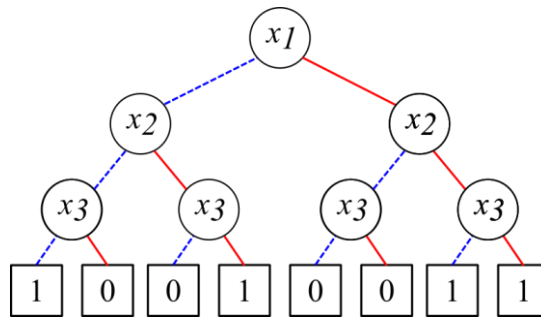


Figure 2 – Binary decision diagram (BDD) representation of a Boolean function.

In order to make the BDD a canonical data structure, Bryant (BRYANT, 1986) proposed a *reduced and ordered binary decision diagram (ROBDD)*. ROBDDs are similar to the representation introduced by (AKERS, 1978), but with a fixed ordering of the decision variables in the graph. Non-terminal nodes controlled by the same variable and pointing for the same left and right child (i.e.  $low(v) = high(v)$ ) are removed. Nodes controlled by the same variable and pointing to the same left child and right child are merged (i.e.  $low(v_i) = low(v_j)$  and  $high(v_i) = high(v_j)$ ). An ROBDD representing the same function of the BDD of Figure 2 is shown in Figure 3. Notice that ROBDD is also a more compact way of representing BDDs.

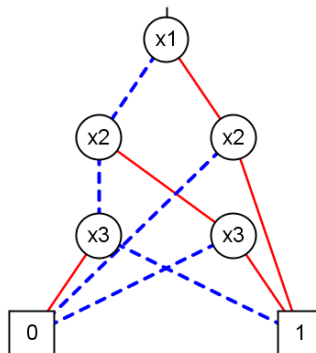


Figure 3 – A reduced and ordered binary decision diagram (ROBDD) of a Boolean function.

### 2.2.1.4 And-Inverter Graphs

An *And-Inverter graph* (AIG) is a directed acyclic graph in which each node has zero or two in-degree (incoming edges). Nodes with no incoming edges are *primary inputs* while the nodes with two incoming edges represent *product (AND) operations*. Edges in an AIG can be *complemented*, meaning the AND operator will use the complemented (inverted) function from such an edge. Notice that nodes can be marked to denote *primary outputs* (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

AIG is a very simple and powerful data structure. Currently is the state-of-the-art data structure for representing large Boolean functions, i.e. functions with few hundred or thousand inputs. Although being a very compact data structure, it lacks canonicity. Figure 4 shows two AIGs that are structurally distinct but represent the same function. Efforts have been made in order to create methods canonize AIGs (MISHCHENKO ET AL, 2013). So far, good heuristic approaches were proposed for canonizing AIGs, but the problem of finding a unique, canonical representation for general AIGs is still open.

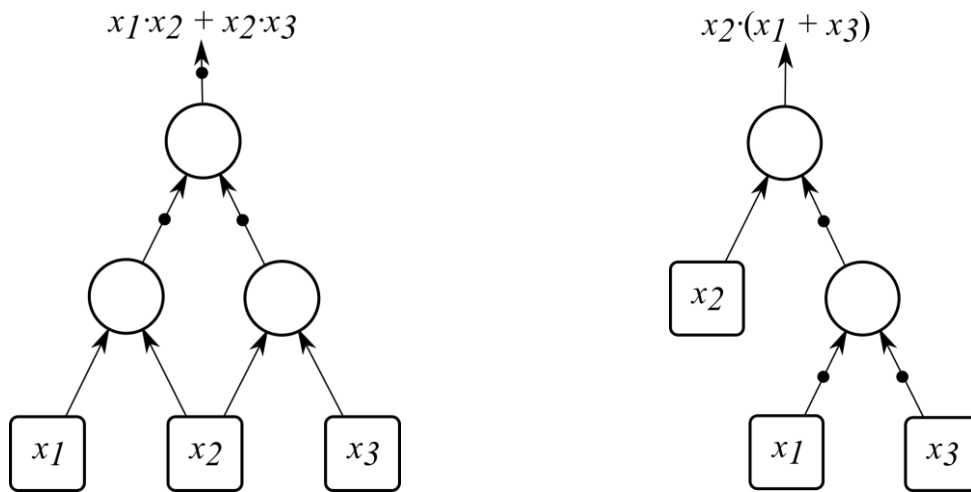


Figure 4 – Example of And-Inverter-Graph (AIG). Complemented edges are denoted by small dots. Notice that primary input nodes  $\{x_1, x_2, x_3\}$  have no incoming edges. Although being structurally distinct, both AIGs represent the same function.

### 2.2.2 Basic operations over Boolean functions

The *complement* (negation, NOT) of a Boolean function  $f$  is a unary operation denoted by  $\bar{f}$  such that  $\text{ON-SET}(\bar{f}) = \text{OFF-SET}(f)$ . A truth table representing the complement operation is presented in Table 2.

Table 2– Truth table representing the complement operation.

$f$	$\overline{f}$
0	1
1	0

The *sum* (union, OR) is a binary operation between two Boolean functions  $f$  and  $g$  denoted by  $f + g$  such that  $\text{ON-SET}(f + g) = \text{ON-SET}(f) \cup \text{ON-SET}(g)$ .

The *product* (intersection, AND) is a binary operation between two Boolean functions  $f$  and  $g$  denoted by  $f \cdot g$  such that  $\text{ON-SET}(f \cdot g) = \text{ON-SET}(f) \cap \text{ON-SET}(g)$ . The product operation will be also represented by juxtaposition, e.g.  $fg$ .

The *exclusive-sum* (difference, exclusive-OR, XOR) is a binary operation between two Boolean functions  $f$  and  $g$  denoted by  $f \oplus g$  such that  $\text{ON-SET}(f \oplus g) = \text{ON-SET}(f + g) \setminus \text{ON-SET}(f \cdot g)$ .

Table 3 summarizes sum, product and exclusive-sum binary operations presented above.

Table 3 – Truth table representing sum, product and exclusive-sum operations over two Boolean functions  $f$  and  $g$ .

$f$	$g$	$f + g$	$f \cdot g$	$f \oplus g$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Given a function  $f(X)$ , the *cofactor operation* (also known as restrict operation) consists of assigning a constant value  $c_i \in \{0, 1\}$  to an input variable  $x_i \in X$ , and is denoted  $f_{x_i=c_i}$  (BOOLE, 1854). The *negative* (*positive*) *cofactor* with respect to (w.r.t) a variable  $x_i$  is denoted by  $f_{x_i=0}$  ( $f_{x_i=1}$ ) or for simplicity  $f_{\overline{x_i}}$  ( $f_{x_i}$ ). In this work the notation  $f(x_0, \dots, x_i = c_i, \dots, x_{n-1})$  is also used to represent a cofactor operation in  $x_i$ . A *cube-cofactor* operation consists of applying cofactors recursively. A cube-cofactor of  $f(X)$  w.r.t the input variables  $x_i, x_j \in X$  is denoted by  $(f_{x_i=c_i})_{x_j=c_j} = f_{x_i=c_i, x_j=c_j}$ . Table 4 shows some cofactors and cube-cofactor examples.



Two distinct input variables  $x_i, x_j \in X$  are *symmetric* in  $f(X)$  if  $f(x_1, \dots, x_i, \dots, x_j, \dots, x_n) = f(x_1, \dots, x_j, \dots, x_i, \dots, x_n)$ . More formally, two input variables are symmetric if and only if  $f_{\bar{x}_i x_j} = f_{x_i \bar{x}_j}$ . In other words, the function  $f$  is unchanged by permuting variable  $x_i$  and  $x_j$ .

Table 4 – Truth table representing every negative and positive cofactor for each input variable. The last column shows the cube-cofactor of  $f$  w.r.t the input variables  $x_1$  and  $x_2$ .

$x_1$	$x_2$	$x_3$	$f$	$f_{\bar{x}_1}$	$f_{x_1}$	$f_{\bar{x}_2}$	$f_{x_2}$	$f_{\bar{x}_3}$	$f_{x_3}$	$f_{x_1 x_2}$
0	0	0	1	1	0	1	0	1	0	1
0	0	1	0	0	0	0	1	1	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	1	1	1	1	0	1	0	1	1
1	0	0	0	1	0	0	1	0	0	1
1	0	1	0	0	0	0	1	0	0	1
1	1	0	1	0	1	0	1	1	1	1
1	1	1	1	1	1	0	1	1	1	1

Some cofactors and cube-cofactors identities are presented as follows. Let  $f(X)$  and  $g(X)$  are two Boolean functions and  $x_i, x_j \in X$ .

$$\overline{(f_{x_i=c_i})} = (\bar{f})_{x_i=c_i} \quad (2)$$

$$(f + g)_{x_i=c_i} = f_{x_i=c_i} + g_{x_i=c_i} \quad (3)$$

$$(f \cdot g)_{x_i=c_i} = f_{x_i=c_i} \cdot g_{x_i=c_i} \quad (4)$$

$$(f \oplus g)_{x_i=c_i} = f_{x_i=c_i} \oplus g_{x_i=c_i} \quad (5)$$

$$f_{x_i=c_i, x_j=c_j} = f_{x_j=c_j, x_i=c_i} \quad (6)$$

$$(x_i)_{x_j} = x_i \quad (7)$$

The *Boolean difference* of a function  $f$  w.r.t a variable  $x_i$  is given by the exclusive-sum of its cofactors:

$$\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\bar{x}_i} \quad (8)$$

### 2.2.3 Unateness analysis

The unateness analysis reveals important information regarding a given Boolean function. It provides details where a Boolean function 1) depends on the information of a particular variable and 2) if it depends on which positive, negative or both polarities is required. The unateness analysis relies on the examination of variable cofactors and their relationship.

We say that a variable  $x_i \in X$  is *redundant* in  $f(X)$  when  $\frac{\partial f(X)}{\partial x_i} = 0$ , which means that the value  $x_i$  does not care when computing  $f$ . Conversely, a variable  $x_i$  belongs to the *support* of  $f$  when its Boolean difference is not zero, i.e.  $\frac{\partial f(X)}{\partial x_i} \neq 0$ . For support (non-redundant) variables, the unateness behavior can be described as follows. The function  $f$  is *positive unate* in  $x_i$  if  $f_{x_i^-} + f_{x_i} = f_{x_i}$ . When  $f_{x_i^-} + f_{x_i} = f_{x_i^-}$ ,  $f$  is *negative unate* in  $x_i$ . We say  $f$  is *binate* in  $x_i$  when  $f_{x_i^-} \neq f_{x_i^-} + f_{x_i} \neq f_{x_i}$ .

When a function  $f$  is either positive or negative unate in all its support variables, we say  $f$  is a unate function. More specifically, if  $f$  is positive (negative) unate in all its support variables, we say  $f$  is a positive (negative) unate function. In the case when at least one variable is binate, the Boolean function is considered binate.

### 2.2.4 Shannon expansion

A Boolean function can be represented through simpler functions. One example is the *Shannon expansion* (also known as Shannon decomposition and Boole's expansion). The Shannon expansion of  $f(X)$  w.r.t a variable  $x_i \in X$  is defined as follows (SHANNON, 1949):

$$f(x_1, \dots, x_i, \dots, x_n) = \bar{x}_i \cdot f_{\bar{x}_i} + x_i \cdot f_{x_i} \quad (9)$$

Eq. 9 presents a sum of simpler functions. We say these functions are simpler since they depend on fewer variables than  $f$ . Notice that a Shannon expansion can also be represented by a product of simpler functions as shown in Eq. 10:

$$f(x_1, \dots, x_i, \dots, x_n) = (\bar{x}_i + f_{x_i}) \cdot (x_i + f_{\bar{x}_i}) \quad (10)$$

## 2.3 Boolean expressions

While algebraic expressions denote mainly numbers, Boolean expressions denote constant truth values false and true, often encoded 0 and 1 respectively. Boolean expressions can be defined recursively as a *constant* **0** or **1**, a variable (*e.g.*  $x_i$ ), or a product, sum or complement of Boolean expressions.

### 2.3.1 Canonical sum-of-products and product-of-sums

The product comprising all support variables of a given Boolean function is called *minterm*. Conversely, the sum of all variables is called *maxterm*. Given a Boolean function  $f(X)$  depending on  $n$  input variables (i.e.  $|X| = n$ ) there are  $2^n$  minterms and, consequently,  $2^n$  maxterms. Table 5 shows, as an example, minterms and maxterms for an arbitrary function depending on three input variables.

A Boolean function  $f$  can be canonically represented by the sum of all minterms  $m_i \in 2^n$  such that  $f(m_i) = 1$ . For example, the function  $f$  presented in Figure 1 has its *canonical sum-of-products* equal to  $F = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} + \overline{x_1} \cdot x_2 \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2 \cdot x_3$  or alternatively  $F = \sum(m_0, m_3, m_6, m_7)$ . Besides, a function can also be represented by the product of its maxterms  $m_i \in 2^n$  such that  $f(m_i) = 0$ . The above-mentioned function  $f$  can be represented by a *canonical product-of-sums*  $F = (x_1 + x_2 + \overline{x_3}) \cdot (x_1 + \overline{x_2} + x_3) \cdot (\overline{x_1} + x_2 + x_3) \cdot (\overline{x_1} + x_2 + \overline{x_3})$  or otherwise as  $F = \prod(m_1, m_2, m_4, m_5)$ .

Notice that a function can be represented by several distinct logic expressions. Usually, some representations are considered better than others. For example, one may need the sum-of-products with the smallest number of cubes (products) as possible. Minimization of Boolean expressions is discussed as follows.

Table 5 – Example of minterms and maxterms enumeration of a function  $f(X)$  (same presented in Figure 1) depending of three input variable  $X = \{x_1, x_2, x_3\}$ .

$m_i \in 2^3$	$x_1$	$x_2$	$x_3$	$f$	<i>minterm</i>	<i>maxterm</i>
$m_0$	0	0	0	1	$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$	$x_1 + x_2 + x_3$
$m_1$	0	0	1	0	$\overline{x_1} \cdot \overline{x_2} \cdot x_3$	$x_1 + x_2 + \overline{x_3}$
$m_2$	0	1	0	0	$\overline{x_1} \cdot x_2 \cdot \overline{x_3}$	$x_1 + \overline{x_2} + \overline{x_3}$
$m_3$	0	1	1	1	$\overline{x_1} \cdot x_2 \cdot x_3$	$x_1 + \overline{x_2} + x_3$
$m_4$	1	0	0	0	$x_1 \cdot \overline{x_2} \cdot \overline{x_3}$	$\overline{x_1} + x_2 + \overline{x_3}$
$m_5$	1	0	1	0	$x_1 \cdot \overline{x_2} \cdot x_3$	$\overline{x_1} + \overline{x_2} + x_3$
$m_6$	1	1	0	1	$x_1 \cdot x_2 \cdot \overline{x_3}$	$\overline{x_1} + x_2 + \overline{x_3}$
$m_7$	1	1	1	1	$x_1 \cdot x_2 \cdot x_3$	$\overline{x_1} + \overline{x_2} + \overline{x_3}$

### 2.3.2 Two-level minimization

Two-level forms are expressions that can be seen as a rooted tree where leaf nodes are variables (or their complement) and the intermediate nodes represent a Boolean operation distinct from the operation performed on the root node. *Sum-of-products* (SOP) is an example of a two-level form where intermediate nodes represent AND operations while the root node represents an OR operation. *Product-of-sums* (POS) is a two-level form with OR as intermediate operations and AND as root operation. The *Exclusive-sum-of-products* (ESOP) has, as root, the XOR operator and AND as intermediate operators.

#### 2.3.2.1 Prime, essential prime and irredundant sum-of-products

When minimizing an SOP, the challenge is to reduce the number of product terms used to represent a given Boolean function. Conversely, for POS minimization, the goal is to reduce the total number of sum terms. Since SOP and POS minimization problems are dual, we will discuss only SOP minimization (BRAYTON ET AL, 1984).

A *cube* is a product of disjoint and possibly complemented variables. A cube containing all input variables  $\{x_0, \dots, x_{n-1}\}$  represents a minterm of  $f(X)$ . A cube  $p$  composed of variables  $\{x_0, \dots, x_{n-1}\}$  is an *implicant* of  $f(X)$  if  $p \Rightarrow f$ , i.e. each assignment that makes  $p$  evaluate to **1** also maps  $f$  to **1**. *Prime implicant* is an implicant which is not contained in any other implicant. For the sake of simplicity, let prime refer to prime implicant in this dissertation. A

prime implicant is an *essential prime implicant* if there is at least one minterm that is covered by that prime, but another prime implicant.

A set of cubes  $S$  is a *cover* for  $f$  if the union (sum) of all cubes in  $S$  represents the function  $f$ . A *minimum cover* is a cover with the minimum number of cubes. A *prime cover* is a cover that consists only of prime implicants. An *irredundant sum-of-products* (ISOP) is a prime cover for  $f$  such that no prime  $p$  in  $S$  can be removed without changing the function the cover represents, i.e.  $S \setminus \{c\} \neq f$ . A given function  $f$  can be represented by several distinct ISOPs (SASAO; BUTLER, 2001). By definition, essential primes of a function  $f$  will be always present in all possible ISOPs of  $f$ .

Consider as an example the function  $f$  presented in Figure 1. Its canonical sum-of-products  $F = \overline{x_1}\overline{x_2}\overline{x_3} + \overline{x_1}x_2x_3 + x_1\overline{x_2}\overline{x_3} + x_1x_2x_3$  contains four cubes. We can try to expand the cube  $\overline{x_1}x_2x_3$  by removing the variable  $x_1$ , generating the cube  $x_2x_3$ . Notice that  $x_2x_3$  is a prime implicant of  $f$ , since both  $\overline{x_1}x_2x_3$  and  $x_1x_2x_3$  results **1** in  $f$  and there is no other cube implicant that contains it. By repeating this process, one can enumerate all primes for  $f$ :  $\{\overline{x_1}x_2, x_2x_3, \overline{x_1}\overline{x_2}\overline{x_3}\}$ . Notice that all three primes are essential, and consequently the minimum cover for  $f$ . For this example, the only ISOP representing  $f$  is  $F = \overline{x_1}x_2 + x_2x_3 + \overline{x_1}\overline{x_2}\overline{x_3}$ . The IPOS representing the same function is  $F = (\overline{x_1} + x_2) \cdot (x_2 + \overline{x_3}) \cdot (x_1 + \overline{x_2} + x_3)$ .

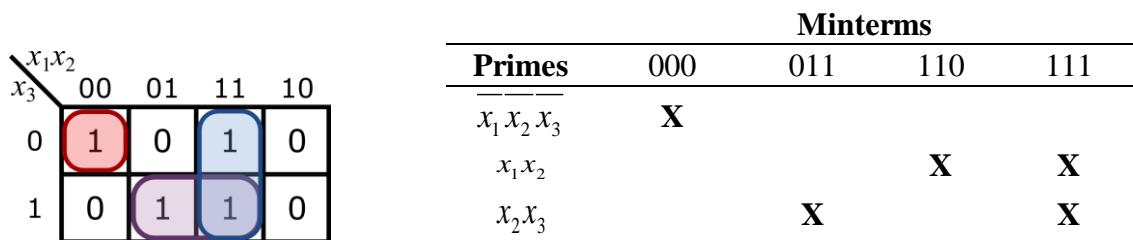


Figure 5 – Karnaugh-map and covering example for  $F = x_1x_2 + x_2x_3 + \overline{x_1}\overline{x_2}\overline{x_3}$ .

There are cases where no essential primes exist for a given function. Consider, for example, the function shown in the Karnaugh-map in Figure 6(left):

$F_2 = \overline{x_1}\overline{x_2}\overline{x_3}x_4 + \overline{x_1}x_2\overline{x_3}x_4 + \overline{x_1}x_2x_3x_4 + x_1\overline{x_2}\overline{x_3}x_4 + x_1\overline{x_2}x_3x_4 + x_1x_2\overline{x_3}x_4 + x_1x_2x_3x_4 + \overline{x_1}x_2\overline{x_3}x_4$ .  
Such function contains eight primes

$\{\overline{x_1 x_2 x_4}, \overline{x_1 x_3 x_4}, \overline{x_1 x_2 x_4}, \overline{x_2 x_3 x_4}, \overline{x_2 x_3 x_4}, \overline{x_1 x_2 x_3}, \overline{x_1 x_3 x_4}, \overline{x_1 x_2 x_3}\}$ . All primes and their respective covered minterms are presented in Figure 6 (right). Notice that all primes are non-essential, i.e. there is no minterm that is covered by one prime only. The minimum covering for such function consists of four primes. The resulting ISOP is  $F_2 = \overline{x_1 x_2 x_4} + \overline{x_2 x_3 x_4} + \overline{x_1 x_2 x_3} + \overline{x_1 x_3 x_4}$ . Notice that more than one minimum solution can be obtained for this example.

		<b>Minterms</b>									
		<b>Primes</b>	0101	0110	0111	1001	1010	1011	1101	1110	
$x_3/x_2$	$x_3 x_4$										
	00	0	0	0	0						
	01	0	1	1	1						
	11	0	1	0	1						
	10	0	1	1	1						
			$\overline{x_1 x_2 x_4}$				X		X		
			$\overline{x_1 x_3 x_4}$				X			X	
			$\overline{x_1 x_2 x_4}$	X		X					
			$\overline{x_2 x_3 x_4}$	X						X	
			$\overline{x_2 x_3 x_4}$		X						X
		$\overline{x_1 x_2 x_3}$		X	X						
		$\overline{x_1 x_3 x_4}$					X			X	
		$\overline{x_1 x_2 x_3}$					X	X			

Figure 6 – Karnaugh-map and a minimum covering example for function

$$F_2 = \overline{x_1 x_2 x_4} + \overline{x_2 x_3 x_4} + \overline{x_1 x_2 x_3} + \overline{x_1 x_3 x_4}.$$

The Two-level SOP minimization problem started to receive attention after the seminal work from Quine (QUINE, 1952), (QUINE, 1955) and McCluskey (MCCLUSKEY, 1956). After that, several minimizer tools were proposed. A non-exhaustive list can include *MINI* (HONG; CAIN; OSTAPKO, 1974), *ESPRESSO* (BRAYTON ET AL, 1984), *Presto-II* (BARTHOLOMEUS; MAN, 1985), *PALMINI* (NGUYEN; PERKOWSKI; GOLDSTEIN, 1987), *ESPRESSO-SIGNATURE* (MCGEER; SANGHAVI; BRAYTON; VINCENTELLI, 1993), *Scherzo* (COUDERT, 1994), and *BOOM* (HLAVICKA; FISER, 2001).

### 2.3.2.2 Exclusive-sum of products

Another well-known canonical representation for Boolean functions is the *canonical exclusive-sum-of-products*. It can be represented by the exclusive-sum of all minterms  $m_i \in 2^n$  such that  $f(m_i) = 1$ , i.e. the same minterms used to compose the canonical sum-of-products. Considering as an example the function presented in Figure 5, the canonical exclusive-sum-of-products for it is  $F = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \oplus \overline{x_1} \cdot x_2 \cdot \overline{x_3} \oplus x_1 \cdot \overline{x_2} \cdot \overline{x_3} \oplus x_1 \cdot x_2 \cdot x_3$ .

The goal of ESOP minimization is to find the ESOP with the minimum number of cubes. For example, the minimum ESOP for the function presented in Figure 5 is  $F = x_2 \oplus \overline{x_1} \cdot \overline{x_3}$ . The resulting cover is shown in Figure 7. Notice that, for the same function, the minimum ISOP contains three cubes (Figure 5) while the minimum ESOP contains two cubes. As already stated by Sasao (SASAO, 1993), ESOPs require, in general, fewer cubes to represent the same function compare to ISOPs forms.

The problem of finding the minimum ESOP is more difficult than finding the minimum ISOP. When minimizing an ISOP, one can use only primes, which are cube implicants. In ESOP minimization, not only cubes implicants but also non-implicant cubes can be used to compose the minimum ESOP. Moreover, in SOP, each minterm must be covered at least once; in ESOP each minterm must be covered an odd number of times. Until nowadays, no efficient method is known to obtain a minimum ESOP except for functions with a small number of inputs (i.e. 8 inputs). There are several heuristic tools to perform ESOP minimization, and among them, we can highlight EXMIN2 (SASAO, 1993), MINT (KOZLOWSKI, 1996) and EXORCISM-4 (MISHCHENKO; PERKOWSKI, 2001).

		$x_1x_2$			
$x_3$		00	01	11	10
0		1	0	1	0
1		0	1	1	0

Figure 7 – ESOP covering resulting  $F = x_2 \oplus \overline{x_1} \cdot \overline{x_3}$ .

### 2.3.3 Factored forms

Examples of two-level forms include SOP, POS, and ESOP. Multi-level forms are expressions with an unbounded number of levels. Clearly, multi-level forms are a super-set of the two-level ones. Factored forms are a subset of multi-level forms. A *factored form* is either a literal or a product or a sum of factored forms, where a *literal* is a variable or its complement (e.g.  $x_i$  or  $\overline{x_i}$ ) (BRAYTON, 1987). For example,  $x_1(x_2 + x_3)$  is a factored form, while  $\overline{x_1(x_2 + x_3)}$  is not; it is, however, a multi-level form.

The goal of factoring algorithms is to provide a minimal literal count expression representing a given Boolean function as input. For example, given  $F_1 = x_0x_2 + x_0x_3 +$

$x_1x_2 + x_1x_3$ , a good factoring algorithm should return  $F_1' = (x_0 + x_1) \cdot (x_2 + x_3)$ . Unfortunately, generating minimum literal factored forms is an NP-hard problem (BRAYTON, 1987). Hence, heuristic algorithms have been developed in order to obtain good factored solutions (SENTOVICH ET AL, 1992), (YOSHIDA; FUJITA, 2011), (MARTINS ET AL, 2012), (MINTZ; GOLUMBIC, 2005). Yet, the quality of the results degrades even for Boolean functions with a small number of inputs, e.g. eight inputs (CALLEGARO, 2012).

## 2.4 Switch networks

A *switch* is a device composed of one control terminal and two contact terminals, where the control terminal determines if there is a connection between the contact terminals. Figure 8(a) depicts a switch controlled by the variable  $x_l$ . Switches can be connected, composing a *switch network*. The function represented by a switch network corresponds to the sum of all simple paths (products) between the contact terminals (SASAO, 1999). A *series association* of switches, as shown in Figure 8(b), represents a conjunction (product, AND) operation, e.g.  $x_1 \cdot x_2$ . A *parallel association* corresponds to a disjunction (sum, OR) operation, e.g.  $x_1 + x_2$ , as depicted in Figure 8(c).

A *series-parallel switch network* (SP) is obtained by iteratively connecting contact terminals in series and/or in parallel. An example of SP network is illustrated in Figure 8 (d), which represents the function  $x_1 \cdot (x_2 + x_3)$ . Notice that it is possible to represent a SP switch network by a factored form where the number of literals on it equal to the number of switches on the network. In this sense, a SP switch network can be transformed directly into a factored form, and vice-versa.

A *non-series-parallel* (NSP) switch network is an arrangement that cannot be achieved by connecting terminal contacts only in series and/or in parallel. An example of an NSP switch network is presented in Figure 8(e), which represents the function  $(x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_5 + x_4 \cdot x_3 \cdot x_2 + x_4 \cdot x_5)$ . Notice that we cannot generate a factored form directly from a NSP switch network. In fact, a minimum factored (and consequently a minimum switch-count SP network) form representing the same function expressed in Figure 8(e) is  $x_1 \cdot (x_2 + x_3 \cdot x_5) + x_4(x_3 \cdot x_2 + x_5)$ , which contains 8 literals (switches).

When constructing switch networks, the goal is to represent a target function using the minimum number of switches. Several methods have been presented in the literature for generating switch networks. Most traditional solutions are based on factoring Boolean



functions, and then using resulting factored forms to directly create series-parallel networks (MINTZ; GOLUMBIC, 2008) (SENTOVICH ET AL, 1992) (MARTINS ET AL, 2012) (MARTINS ET AL, 2010). On the other hand, there are graph-based methods that are able to find both SP and NSP arrangements. Methods that explore NSP arrangements are superior to the ones that create SP only, since they potentially use fewer switches to represent the same functions (ZHU; ABD-EL-BARR, 1993), (KAGARIS; HANIOTAKIS, 2007), (DA ROSA ETL AL, 2007), (POSSANI ET AL, 2012), (POSSANI ET AL, 2016).

Another important concern when generating switch networks is the maximum number of switches in series. In some design styles, like CMOS, the rule-of-thumb is that there should be at most four transistors in series per logic gate (single stage). Some Boolean functions cannot be implemented with such a restriction, e.g. and NAND gate with 5 inputs. The Minimum Decision Chain (MDC) is a property of a Boolean function that results the maximum number of switches in series that is necessary to implement such a function (MARTINS ET AL, 2011). In order to achieve high-performance design, switch networks should not have more transistors in series than the MDC of the function they implement.

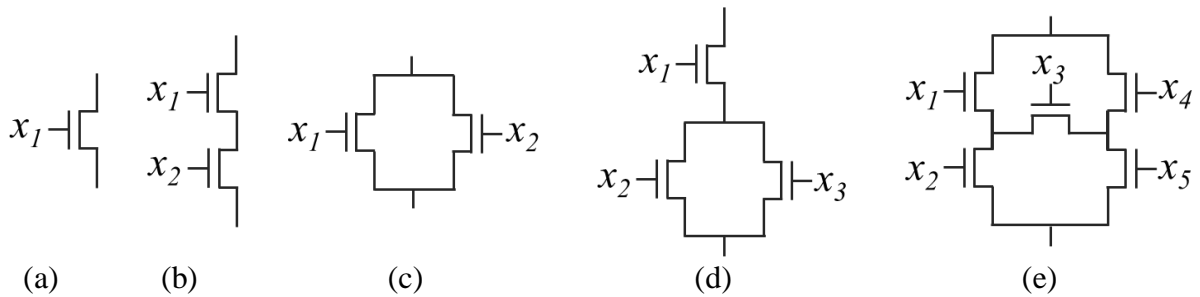


Figure 8 – Examples of switch networks: a) a switch controlled by the variable  $x_1$  b) a series association representing  $x_1 \cdot x_2$  c) a parallel association representing  $x_1 + x_2$  d) a series-parallel network, created by series and parallel associations representing  $x_1 \cdot (x_2 + x_3)$  and e) a non-series-parallel network representing  $(x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_5 + x_4 \cdot x_3 \cdot x_2 + x_4 \cdot x_5)$ .

### 3 CLASSES OF BOOLEAN FUNCTIONS

There are several ways of classifying Boolean functions. Indeed, there are an unlimited number of possible distinct classes. However, one can create a class of Boolean functions in order to explore some intrinsic and interesting propriety of such a class. Usually, the goal is to take advantage of such properties in order to optimize some arbitrary cost. Some useful classes of Boolean functions are presented as follows.

#### 3.1 Unate classes of functions

As presented in Section 2.2.3 – Unateness analysis – an input variable can be redundant, positive unate, negative unate or binate in  $f$ . A function is constant if all its inputs are redundant. A function  $f$  is considered unate if all its input variables are unate in  $f$ . When at least one variable is binate, the function is considered binate.

The properties of unate functions were explored in several scenarios. For example, if a function  $f$  is unate, a resulting ISOP representing  $f$  will have each positive (negative) unate variable appearing exclusively as a positive (negative) literal. Moreover, in unate functions all primes are essential, meaning that the solution for the minimum set covering step is trivial (MCCLUSKEY, 1956). Furthermore, given a unate SOP with  $m$  cubes as input of a two-level minimizer, the prime enumeration and set covering steps are not necessary; the single cube containment method is enough to obtain the minimum ISOP in a time complexity of  $O(m^2)$  (BRAYTON ET AL, 1984).

Another important property of unate functions can be explored in the Boolean satisfiability problem (SAT). The SAT problem is to determine if there exists an interpretation that satisfies a Boolean formula. In other words, is there a variable assignment such that the formula evaluates to **1**. SAT is one of the first problems to be proved NP-Complete (COOK, 1971). However, the problem of satisfying a formula  $F$  is trivial if  $f$  is a unate

function. Each positive (negative) variable in  $f$  must be assigned to 1 (0) in order to make the formula evaluates to **1**.

### 3.2 Symmetric Boolean functions

Two distinct input variables  $x_i, x_j \in X$  are symmetric in  $f(X)$  if  $f(x_1, \dots, x_i, \dots, x_j, \dots, x_n) = f(x_1, \dots, x_j, \dots, x_i, \dots, x_n)$ . In other words, the function  $f$  is unchanged by permuting variable  $x_i$  and  $x_j$ . A function is totally symmetric if is invariant under any permutation of its variables (CHRZANOWSKA-JESKE, 1999). Examples of totally symmetric functions are sum, product, exclusive-sum, majority (voter) functions, etc. Well-known examples of non-symmetric functions include the multiplexer  $F = \bar{x}_1 \cdot x_2 + x_1 \cdot x_3$ , implication  $F = \bar{x}_1 + x_2$  and sharp (inhibition)  $F = \bar{x}_1 \cdot x_2$  functions.

A totally symmetric function  $f$  can be canonically identified by its Shannon set. Let  $f$  depend on  $n$  input variables. A Shannon set  $S_{a_1, \dots, a_k}(x_1, \dots, x_n)$  comprises a set of integers  $\{a_1, \dots, a_k\}$ , where  $0 \leq a_i \leq n$ , such that  $f = \mathbf{1}$  when and only when  $a_i$  of the variables have a value of 1 (SHANNON, 1938). For example,  $S_{1,2}(x_1, x_2)$  represents an OR function of two inputs, where at least one or both variables should be equal 1 to make the OR function evaluate to **1**. An AND function of two inputs is represented by  $S_2(x_1, x_2)$ , while an XOR as is denoted by  $S_1(x_1, x_2)$  and the majority function depending on three inputs is represented by  $S_{2,3}(x_1, x_2, x_3)$ . By representing a function by its Shannon sets, properties like small memory consumption for representation and function's periodicity are explored in the context of cryptographic analysis (CANTEAUT; VIDEAU, 2005).

Totally symmetric functions have several important properties. For example, Shannon in (SHANNON 1938) showed that totally symmetric functions have circuit complexity of at most  $n^2$ . Bryant in (BRYANT, 1986) revealed that BDDs representing totally symmetric functions has at most  $n^2$  nodes. For more information about symmetric functions, (ZHANG ET AL, 2016) is suggested.

### 3.3 Read-Once Boolean functions

A factored form is called *read-once* (RO) if each variable appears only once. A Boolean function is RO if it can be represented by an RO form (HAYES, 1975). For example, the

Boolean function represented by  $F = x_1x_2 + x_1x_3x_4 + x_1x_3x_5$  is a RO function, since it can be factored into  $F = x_1(x_2 + x_3(x_4 + x_5))$ .

If a given function  $f$  can be factored into an RO form, then all input variables are either positive or negative unate in  $f$  (HAYES, 1975). This is a necessary but not sufficient condition since there are unate functions that cannot be factored into an RO form. For example, the unate function  $F = x_1 \cdot x_2 + x_1 \cdot x_3 + x_2 \cdot x_3$  has  $F = x_1 \cdot (x_2 + x_3) + x_2 \cdot x_3$  as the minimal solution, in which variables  $x_2$  and  $x_3$  appear more than once. Functions with binate variables are not RO functions since both positive and negative literals of binate variables should appear in the minimum factored form, i.e. each binate variable will appear at least twice.

The classes of read-once functions have interesting special properties (KARCHMER ET AL., 1993; BOROS; GURVICH; HAMMER, 1994; BOROS; IBARAKI; MAKINO, 1998; GOLUMBIC, 2004). The class of read-once functions is of special interest in several areas, including learning theory (ANGLUIN; HELLERSTEIN; KARPINSKI, 1993; BSHOUTY; HANCOCK; HELLERSTEIN, 1995), databases (SEN; DESHPANDE; GETOOR, 2010) (KANAGAL; LI; DESHPANDE, 2011), digital circuit design (HAYES, 1975), (PEER; PINTER, 1995) and test (HAYES, 1971).

Read-once functions can be implemented using a linear number of AND and OR gates, while most of Boolean functions require an exponential number of gates regarding number of inputs to be implemented (SHANNON, 1949). Besides, RO functions of  $n$  inputs can be implemented by series-parallel switch networks depending on  $n$  switches. The network can be obtained by applying series and parallel expansions for each AND and OR operations on the RO factored form, respectively. For example, the rooted tree and its respective switch network representing  $F = x_1(x_2 + x_3(x_4 + x_5))$  is presented in Figure 9.

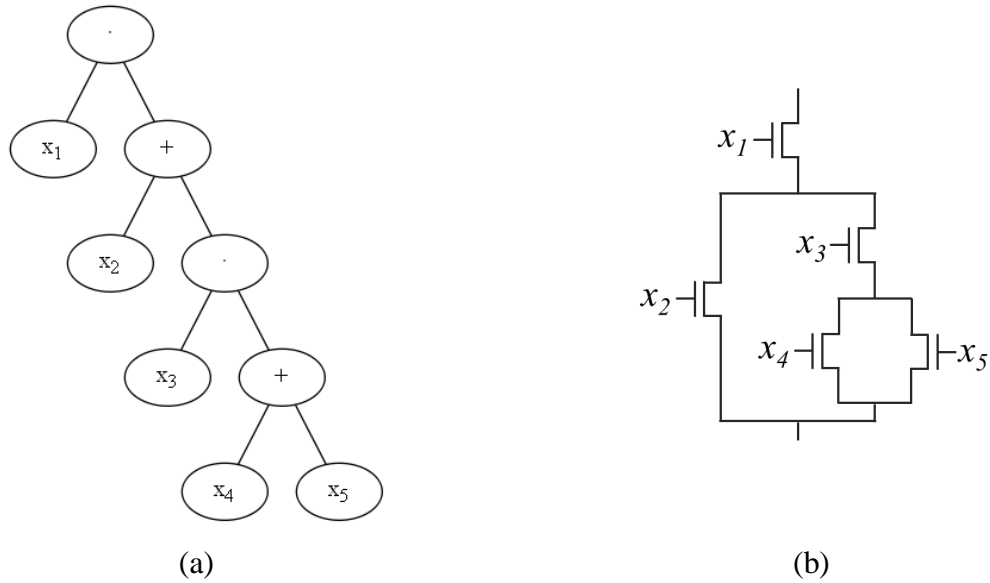


Figure 9 – A rooted tree (a) and its respective series-parallel switch network (b) representing the RO function  $F = x_1(x_2 + x_3(x_4 + x_5))$ .

### 3.4 Read-Polarity-Once Boolean functions

A *read-polarity-once* (RPO) form is a factored form where each polarity (positive or negative) of a variable appears at most once. A Boolean function is RPO if it can be represented by an RPO factored form (CALLEGARO ET AL, 2012), (CALLEGARO ET AL, 2014). For example the function  $F = \bar{x}_1x_2x_4 + x_1x_3 + x_2x_3$  is RPO, since it can be factored into  $F = (\bar{x}_1x_4 + x_3)(x_1 + x_2)$ .

Although the definition of RPO is similar to the read-once forms, the number of functions that arise in the RPO class is much higher than RO. Indeed, by definition, the set of read-once functions is a subset of the RPOs. Unlike read-once, RPO functions can represent not only unate but also binate functions. Among all 2-input functions, only the XOR ( $\bar{x}_1x_2 + x_1\bar{x}_2$ ) and XNOR ( $x_1x_2 + \bar{x}_1\bar{x}_2$ ) functions are not read-once but are RPO.

An RPO expression representing a function  $f$  can be proved as minimum literal form if each unate variable in  $f$  contributes exactly one literal and each binate variable in  $f$  contributes exactly two literals (one positive and one negative) in the RPO form. Let  $f$  be a Boolean function defined by  $F = \bar{x}_1x_2x_4 + x_1x_3 + x_2x_3$ . The variable ‘ $x_1$ ’ is binate in  $f$ , while variables  $\{x_2, x_3, x_4\}$  are positive unate. The RPO factored form  $F = (\bar{x}_1x_4 + x_3)(x_1 + x_2)$  can be proved as minimum factored, since the binate variable ‘ $x_1$ ’ appears twice (once as

positive and once as negative literal), whereas each unate variable  $\{x_2, x_3, x_4\}$  appears only once.

Similarly to read-once, RPO functions can be implemented using a linear number of AND and OR gates. RPO functions of  $n$  inputs can be implemented by series-parallel switch networks depending on at most  $2n$  switches. Indeed, the number of switches can be calculated as follows. Let  $f$  be an RPO function depending on  $n$ -inputs, where  $\alpha$  of these inputs are unate and  $\beta$  are binate variables (i.e.  $\alpha + \beta = n$ ). The number of switches necessary to represent an RPO function is  $\alpha + 2\beta$ . For example,  $F = (\overline{x_1}x_4 + x_3)(x_1 + x_2)$  is an RPO function with  $\alpha = 3$  and  $\beta = 1$  and can be implemented using 5 switches. The rooted tree and its respective switch network representing  $F$  is presented in Figure 10.

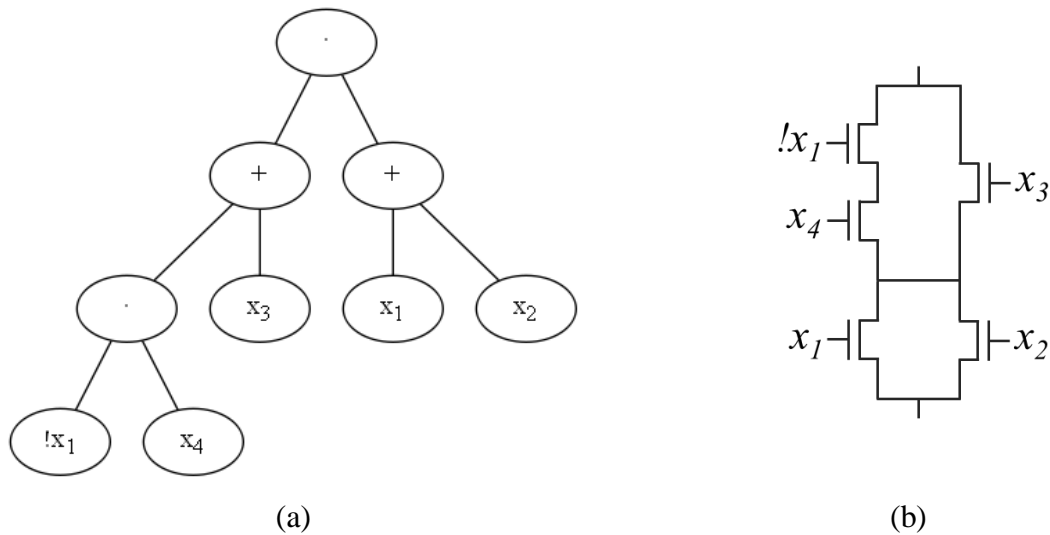


Figure 10 – Rooted tree (a) and its respective series-parallel switch (b) network representing the RPO function  $F = (\overline{x_1}x_4 + x_3)(x_1 + x_2)$ .

### 3.5 Boolean decomposition

Decomposition is the task of representing complex Boolean functions through simpler subfunctions (ASHENHURST, 1957), (CURTIS, 1962). A Boolean function  $f(X)$  can be expressed through subfunctions  $g$  and  $h$ , such that:

$$f(X) = h(g(X_1), X_2) \tag{11}$$

where  $X_1, X_2 \neq \emptyset$ , and  $X_1 \cup X_2 = X$ . If such a representation exists, it is considered a *functional decomposition* of  $f$ . Functions  $g$  and  $h$  are named *predecessor* and *successor*

functions,  $X_1$  and  $X_2$  are called *bound-set* and *free-set*, respectively (ASHENHURST, 1957), (CURTIS, 1962).

There are several classifications regarding decomposition types. For example, in Ashenhurst decomposition, each block (subfunction) must have one single output; while in Curtis decomposition blocks can have multiple outputs. A *disjoint-support decomposition* (DSD) is a special case of functional decomposition, where the input sets  $X_1$  and  $X_2$  do not share any element, *i.e.*,  $X_1 \cap X_2 = \emptyset$ . When blocks share inputs then the decomposition is named *non-disjoint*.

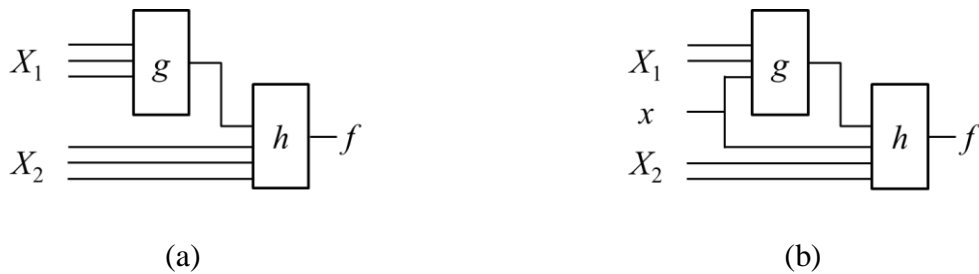


Figure 11 – Schematic of (a) disjoint and (b) non-disjoint decompositions.

SOURCE: (SASAO; BUTLER, 1997).

In general, the worst case number of transistors needed to implement a function depending on  $n$  input variables is  $2^n/n$  (SHANNON, 1949). In this sense, disjoint-support decompositions play a major role on gate minimization since they greatly reduce the number of inputs of each block. For example, suppose that an  $n$ -input function  $f$  can be decomposed as depicted in Figure 11 (a). Let  $n_1 = |X_1|$  and  $n_2 = |X_2|$ , such that  $n = n_1 + n_2$ . Therefore, blocks  $g$  and  $h$  can be implemented using at most  $2^{n_1}/n_1$  and  $2^{n_2+1}/(n_2 + 1)$  gates, respectively. For a large number of inputs,  $2^n/n \gg 2^{n_1}/n_1 + 2^{n_2+1}/(n_2 + 1)$  (SASAO, 1998).

Decompositions where each block has two or fewer inputs are called bi-decomposition. Let  $f$  be bi-decomposed as  $f(X) = h(g_1(X_1 \cup X_3), g_2(X_2 \cup X_3))$  and  $X_1$ ,  $X_2$  and  $X_3$  be disjoint. If  $X_3$  is empty, the decomposition is disjoint. When  $X_1$ ,  $X_2$  and  $X_3$  are non-empty, the decomposition is called strong. When  $X_1$  or  $X_2$  is empty the decomposition is called weak (SASAO; BUTLER, 1997). Figure 12 depicts the schematic for (a) disjoint, (b) strong and (c) weak bi-decompositions.

A *full disjoint-support bi-decomposition* is a decomposition tree where all blocks from the output to the primary inputs are disjoint-support bi-decompositions. Full disjoint-support bi-decompositions are of special interest since the number of gates necessary to implement them grows linearly with the number of inputs (BERTACCO; DAMIANI, 1997). Disjoint-support decompositions have been applied to different IC design domains including ASIC and FPGA design, synthesis, placement, routing, and verification (KARPLUS, 1990), (SASAO, 1981), (MURGAI ET AL, 1990), (KUTZSCHEBAUCH; STOK, 2002), (BERTACCO; OLUKOTUN, 2002), (PLAZA; BERTACCO, 2005).

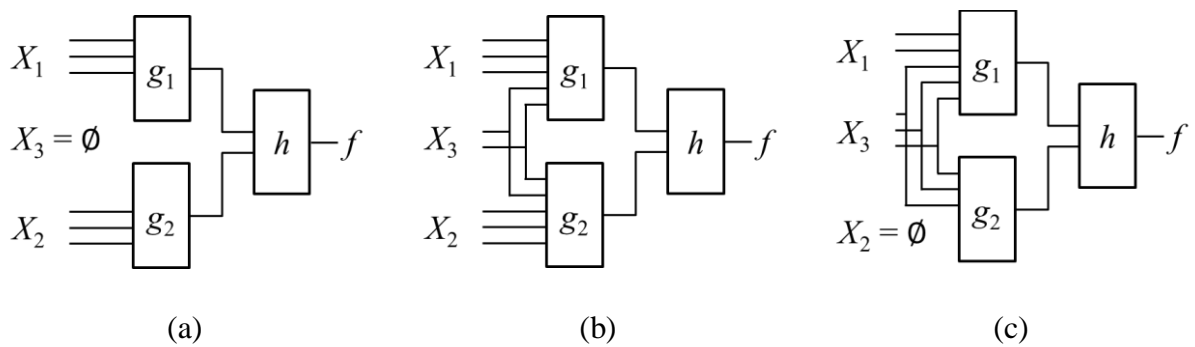


Figure 12 – Schematic examples of (a) disjoint, (b) strong and (c) weak bi-decompositions.

SOURCE: (MISHCHENKO; STEINBACH; PERKOWSKI, 2001).

### 3.6 Comparison of Boolean function classes

This section presents some classes of Boolean functions that are related to the subject of this dissertation. In the following, a comparison about RO, DSD and RPO will be performed. These three classes are closely related since RO class is a subset of both DSD and RPO classes. Figure 13 depicts the relationship among unate, RO, DSD and RPO classes. Notice that all functions that are DSD and unate are also RO. The same property is observed for RPO functions. Besides, there are several functions that are classified both DSD and RPO.



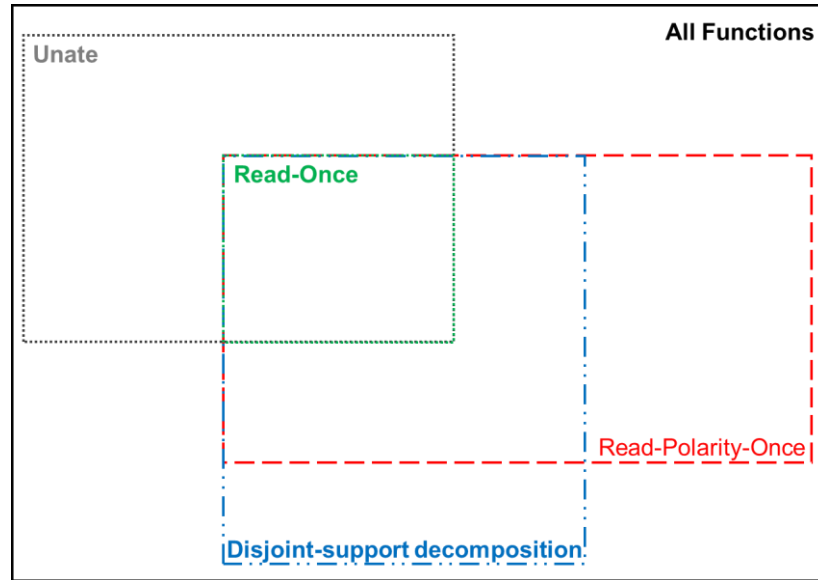


Figure 13 – Comparison of unate, read-once (RO), full disjoint-support bi-decomposition (DSD) and read-polarity-once (RPO) classes. As can be seen, the class of RO functions is a subset of both DSD and RPO classes.

In (HAYES, 1976), an enumeration of all RO functions up to 8 inputs was carried out. Results of this evaluation are shown in Table 6, second column. In (BUTLER, 1975), an enumeration of DSD functions up to 15 inputs was performed. The DSD data (limited herein to 8 inputs) is presented in Table 6, third column. In (KODANDAPANI; SETH, 1978), an enumeration of both RO and DSD functions up to 7 inputs was performed. Both numbers of RO and DSD match with Hayes’ and Butler’s results.

In (CALLEGARO ET AL, 2012), an enumeration of RPO functions up to 5 inputs was carried out. The same authors recently extended the enumeration for functions with up to 6 inputs. The number of RPO functions is shown in Table 6, fourth column. The fifth column shows the number of functions that belong to both DSD and RPO classes of functions. Each line in Table 6 represents the total number of functions per class with up to  $n$  inputs, where  $2 \leq n \leq 8$ . The number of all possible functions with  $n$  inputs is  $2^{2^n}$ .

Table 6 – Enumeration of classes of functions: read-once (RO), full disjoint-support bi-decomposition (DSD) and read-polarity-once (RPO).

Inputs	RO	DSD	RPO	DSD $\cap$ RPO	$\frac{DSD \cap RPO}{DSD}$
2	12	14	14	14	100%
3	94	150	228	148	99%
4	1,144	2,678	20,748	2,492	93%
5	19,994	68,966	6,814,286	57,894	84%
6	456,774	2,311,640	3,934,102,220	1,699,626	74%
7	12,851,768	95,193,064	-	-	-
8	429,005,426	4,645,069,336	-	-	-

## 4 READ-ONCE FUNCTIONS

The class of read-once functions has interesting special properties (KARCHMER ET AL., 1993; BOROS; GURVICH; HAMMER, 1994; BOROS; IBARAKI; MAKINO, 1998; GOLUMBIC, 2004). The class of read-once functions is of special interest in several areas, including learning theory (ANGLUIN; HELLERSTEIN; KARPINSKI, 1993; BSHOUTY; HANCOCK; HELLERSTEIN, 1995), databases (SEN; DESHPANDE; GETOOR, 2010) (KANAGAL; LI; DESHPANDE, 2011) and digital circuit design (PEER; PINTER, 1995).

In this chapter, a synthesis method that finds, whenever it is possible, a read-once realization for a target function is presented. The method relies on the fact that the class of RO functions is closed under cofactor operations, i.e. a cofactor of an RO function is an RO function (HAYES, 1975). Given a function  $f(X)$  depending on  $n$  inputs and a variable  $x_i \in X$ , the idea is to obtain recursively read-once trees for negative and positive cofactors w.r.t. variable  $x_i$  and, based on these two trees, decide where to insert  $x_i$  in one of these trees.

### 4.1 Previous work

The class of read-once (RO) Boolean functions has been known for a long time: it was first introduced by Hayes (HAYES, 1975) who called them *fanout-free functions*. The method proposed by Hayes suffers from high complexity since the algorithm makes intensive calls to a procedure to perform equivalence checking of cofactors.

Peer and Pinter (PEER; PINTER, 1995) also proposed an algorithm to synthesize *non-repeating literal trees*, another name to *read-once* functions. The main drawback of their method is that it runs in non-polynomial time. The main reason is that their method requires intensive SOP to POS (and POS to SOP) conversions which require an exponential time to run, making the method very costly in runtime.

More recent work was proposed in order to overcome the limitations of Hayes' and Peer and Pinter's methods. Golumbic (GOLUMBIC; MINTZ; ROTICS, 2001) was the first to propose a polynomial time factoring algorithm for RO functions, called IROF. His method is based on Gurvich's work (GURVICH, 1991). Another recent work was proposed by Lee (LEE; WANG, 2007) and is based on Hayes' work. His method replaced the equivalence checking of cofactors by a property called disappearance, turning the algorithm feasible in polynomial time.

## 4.2 Proposed method for Read-Once synthesis

A synthesis method that finds, whenever it is possible, a read-once realization for a target function is proposed. The method was designed based on a divide-and-conquer strategy. Finding a read-once tree for a target function consists of obtaining read-once trees for simpler sub-problems: negative and positive cofactors (division phase). These solutions are then composed (conquer phase), resulting in a read-once solution for the original problem (target function). The method is independent of the Boolean function's data structure representation. It relies only on cofactor operation and equivalence checking regarding constants.

### 4.2.1 Notation and definitions

**Definition 1:** A *read-once tree* is a constant node ( $\mathbf{0}$  or  $\mathbf{1}$ ), a literal ( $x_i$  or  $\bar{x}_i$ ) node or a sum (+) or product ( $\cdot$ ) node of (non-constant) read-once trees with disjoint support.

**Definition 2:** An *operator node*  $n_i$  can be either a sum or a product node, and  $\text{op}(n_i)$  is a function that returns "+" or " $\cdot$ " if  $n_i$  is a sum or a product node, respectively.

**Definition 3:** The function neighbor  $N(n_i)$  can be applied over operator nodes and returns a list of  $n_i$  adjacent (child) nodes.

**Definition 4:** A read-once tree is said to be *collapsed* if there is no sum (product) node containing a sum (product) node as child, respectively.

**Definition 5:** A *collapse operation* over a tree  $T$  consists of finding an operator node  $n_i$  that has an operator child  $n_j$  such that  $\text{op}(n_i) = \text{op}(n_j)$ , and adding all  $N(n_j)$  into the  $n_i$  child node list, i.e.  $N(n_i) = N(n_i) \cup N(n_j)$ . After that, node  $n_j$  is deleted. The collapse operation runs until no more non-collapsed nodes are found, resulting in a collapsed tree. Figure 14 (left) shows a non-collapsed tree and its collapsed version (right).

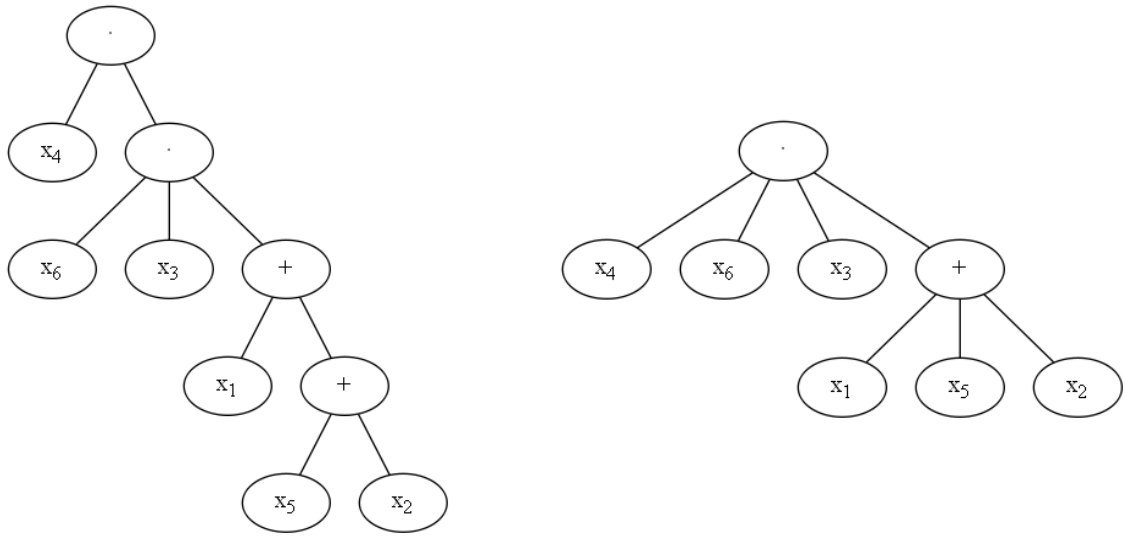


Figure 14 – Example of a non-collapsed tree (left) and the resulting tree after the collapse operation (right).

**Definition 6:** A *sort operation* over a tree  $T$ , given an input order  $Y = [y_1, \dots, y_i, y_{i+1}, \dots, y_n]$ , where  $y_i \in X$ , orders subtrees of  $T$  regarding their priority on  $Y$ . Constant nodes have priority over all other nodes, i.e.  $0 \leq 1 \leq Y$ . Each subtree has priority equal to the highest priority of its subtrees. Consider the tree depicted in Figure 15 (left). A sorted tree with an input order  $Y = [x_1, x_2, x_3, x_4, x_5, x_6]$  is presented in Figure 15 (right).

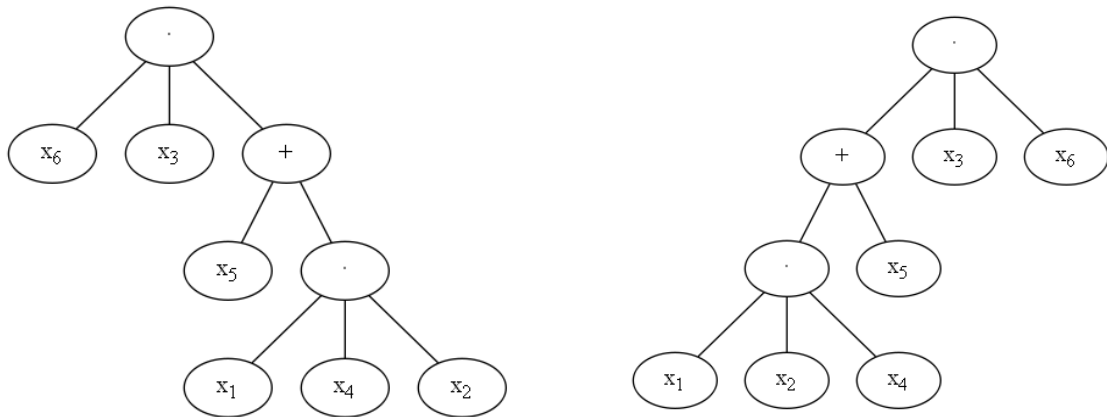


Figure 15 – Example of sorting trees. Original tree (left) and sorted tree with input order  $Y = [x_1, x_2, x_3, x_4, x_5, x_6]$  (right).

**Theorem 1:** Given a fixed input order, i.e.  $Y = [x_1, \dots, x_n]$ , collapsed and sorted read-once trees are *canonical* representations.

**Proof:** (MISHCHENKO; BRAYTON, 2013) ■

**Definition 7:** The *support of a tree*  $\text{sup}(T)$  is a set containing all variables appearing on  $T$ . For example, let  $T = x_1 + \bar{x}_2(x_3 + \bar{x}_4)$ , then  $\text{sup}(f) = \{x_1, x_2, x_3, x_4\}$ . Notice that the support of a tree representing a constant node is empty, i.e.  $\text{sup}(0) = \text{sup}(1) = \emptyset$ .

**Definition 8:** *Matching* between trees can be recursively defined as follows. Two trees  $T_1$  and  $T_2$  are considered a *match* if both trees represent the same constant or represent the same literal or both  $T_1$  and  $T_2$  are operator nodes such that  $\text{op}(T_1) = \text{op}(T_2)$ ,  $|N(T_1)| = |N(T_2)| = m$  and all descendants from  $T_1$  match, in order, descendants from  $T_2$ , i.e.  $\text{match}(N(T_1)_i, N(T_2)_i)$ , for  $1 \leq i \leq m$ .

**Theorem 2:** Two canonical (collapsed and sorted) read-once trees represent the same Boolean function if and only if the trees match.

**Proof:** Straightforward from Theorem 1 and definition 8. ■

#### 4.2.2 Read-once by cofactor composition

A method for synthesis of read-once functions is proposed. The method relies on the fact that the class of RO functions is closed under cofactor operations, i.e. a cofactor of an RO function is an RO function. For example, Table 7 shows an enumeration of all possible read-once trees regarding an arbitrary variable  $x_i$ 's position. For each case, a read-once tree is depicted in column  $F$ , negative and positive cofactors w.r.t input  $x_i$  are shown in column  $F_{x_i=0}$  and  $F_{x_i=1}$ , respectively. Nodes depicted by triangles denote read-once subtrees.

In total, 20 cases of interest are shown in Table 7. A target variable  $x_i$  can be redundant in the tree  $T$ , i.e.  $x_i \notin \text{sup}(T)$ , represented by the cases 1 and 2. Cases 3 and 4 represent the case where  $T$  is composed by a literal of variable  $x_i$ . Cases 5-12 show examples where variable  $x_i$  is a child of an operator node, which is positioned in the root of  $T$ . Cases 13-20 represent the trees where  $x_i$  is a child of an operator node that is not the root node. In this sense, Table 7 virtually enumerates all possibilities of read-once trees regarding the observation of an arbitrary variable  $x_i$ . Notice that, in cases 13-20, part of the read-once tree is hidden, without lack of generality and just for presentation sake, as detailed in Figure 16.

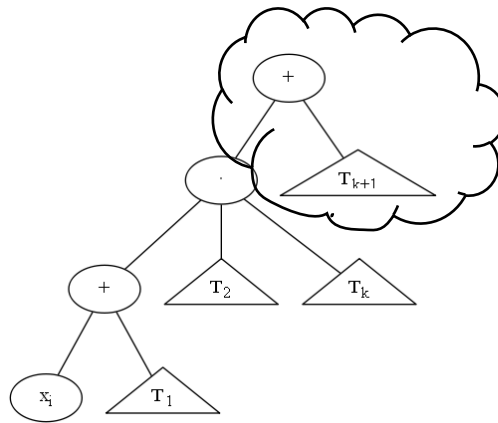












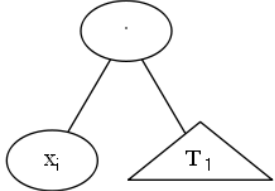

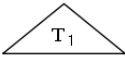
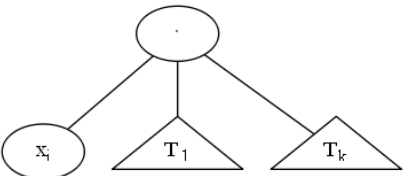

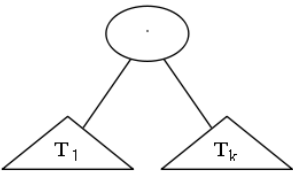


Figure 16 – Example explicitly showing the hidden part from 13-20 (Table 7). The top part (cloud) of the tree is hidden without lack of generality and just for presentation sake.

Table 7 – Enumeration of all possible read-once trees regarding variable  $x_i$ 's position. For each case, a read-once tree is depicted in column  $F$ , and negative and positive cofactors w.r.t input  $x_i$  are shown in column  $F_{x_i=0}$  and  $F_{x_i=1}$ , respectively. Nodes depicted by triangles denote read-once subtrees. Cases 1-13 represent the complete read-once tree. In cases 13-20, part of the read-once tree is hidden, without lack of generalization. The table shows 20 cases of interest.

Case

ID	$F$	$F_{x_i=0}$	$F_{x_i=1}$
1			
2			
3			
4			
5			
6			



Case

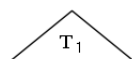
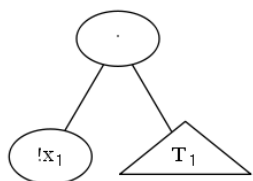
ID

$F$

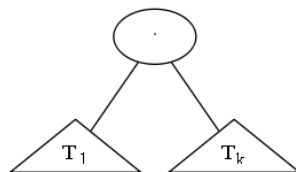
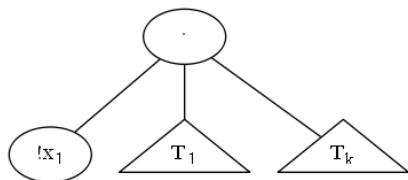
$F_{x_i=0}$

$F_{x_i=1}$

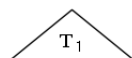
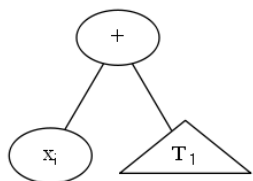
7



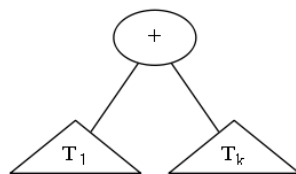
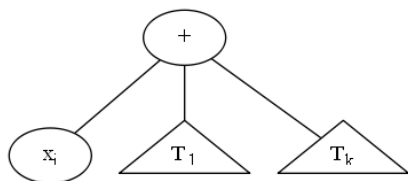
8



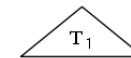
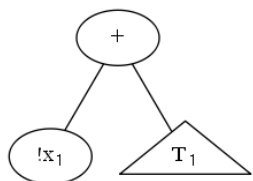
9



10

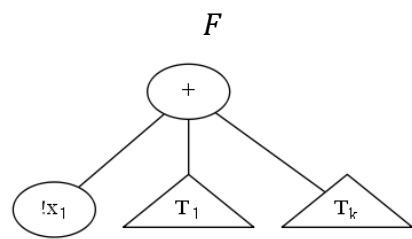


11

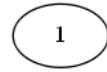
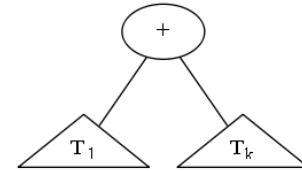


Case

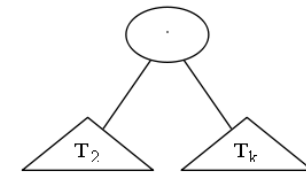
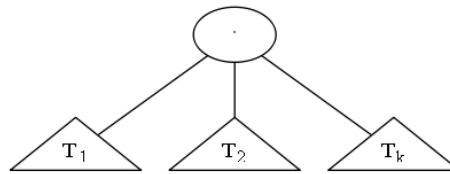
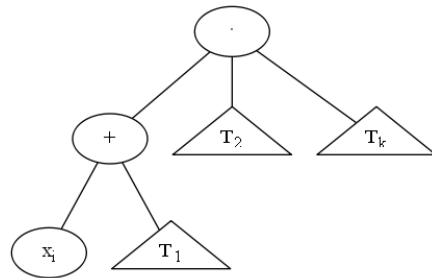
ID



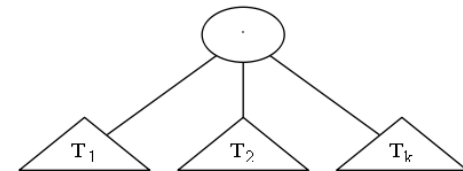
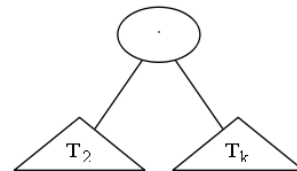
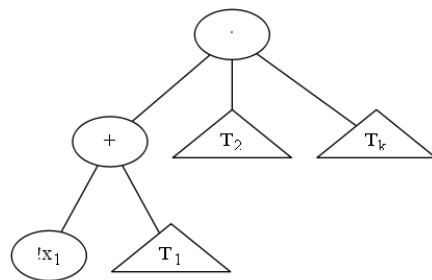
12

 $F_{x_i=0}$  $F_{x_i=1}$ 

13



14



Case

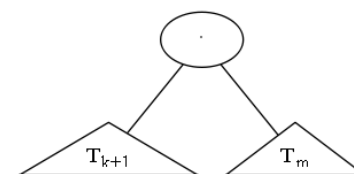
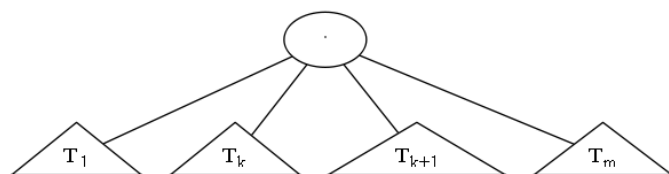
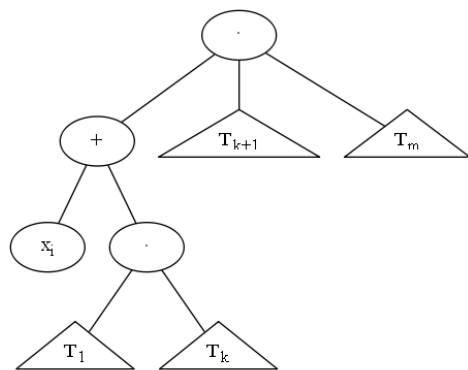
ID

$F$

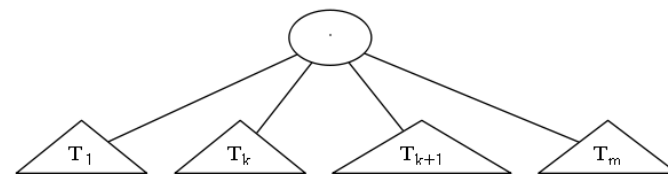
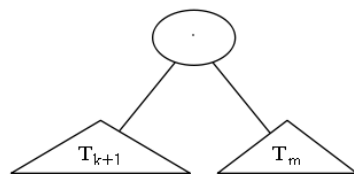
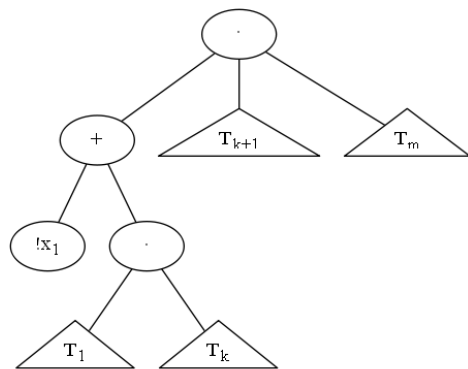
$F_{x_i=0}$

$F_{x_i=1}$

15

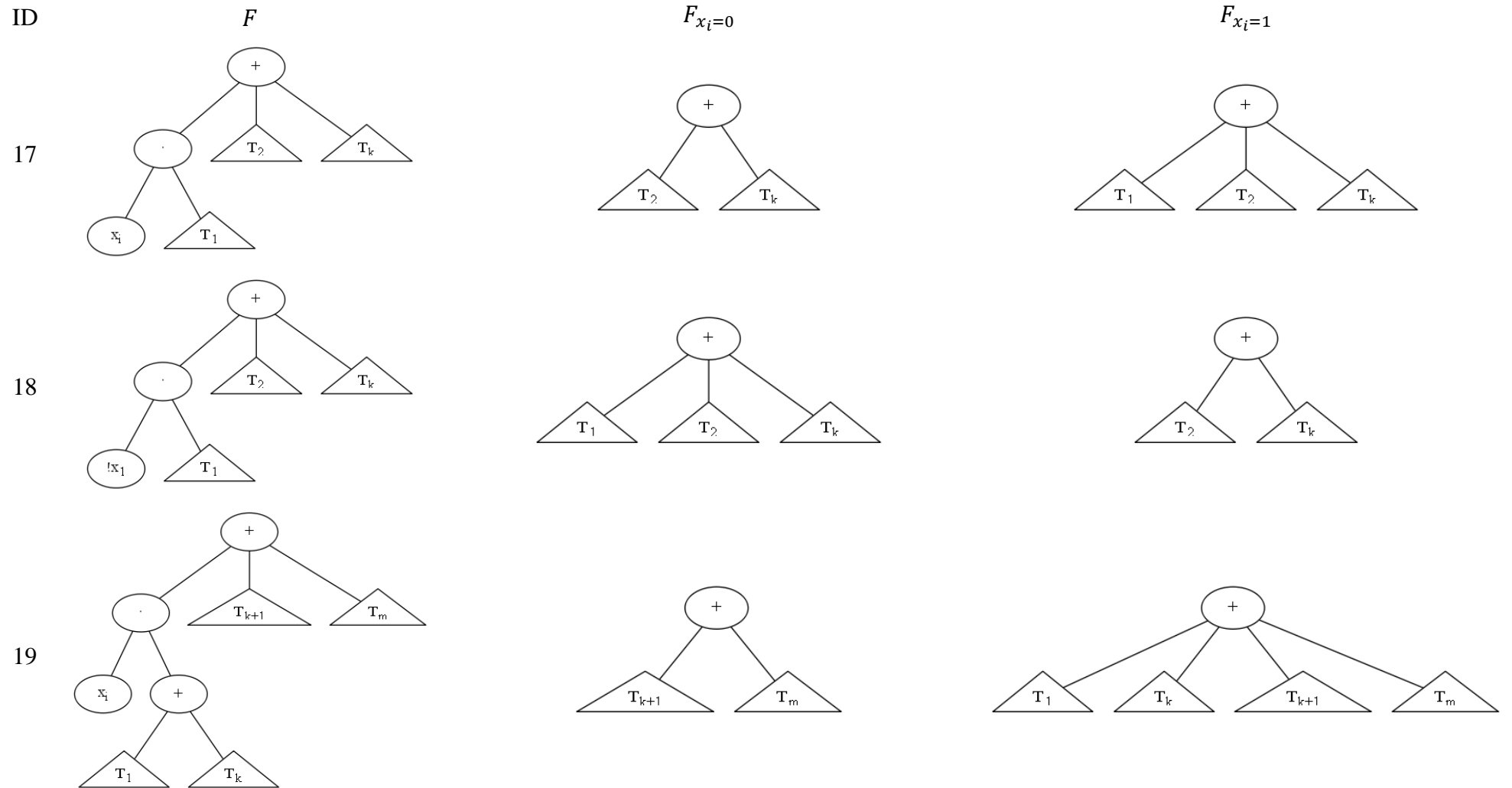


16



Case

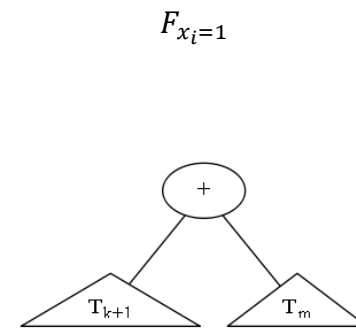
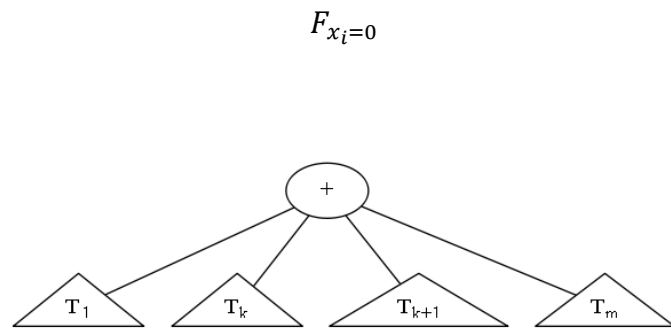
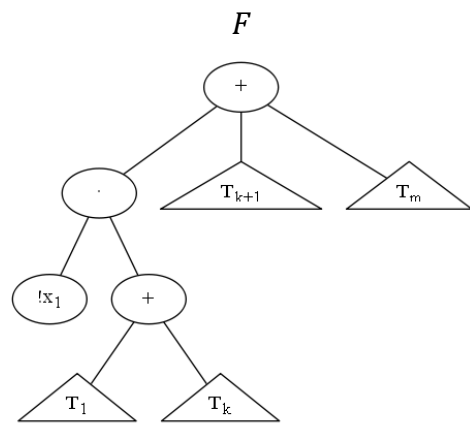
ID



Case

ID

20



Given a function  $f(X)$  depending on  $n$  inputs and a variable  $x_i \in X$ , the idea is to obtain recursively read-once trees for negative and positive cofactors w.r.t.  $x_i$  and based on these two trees, decide whenever possible insert  $x_i$  in the one of these trees. As shown in Table 7, if  $f$  is a read-once function, both negative and positive cofactors  $x_i$  should also be read-once functions. This is the working principle of the proposed **RO\_BY\_COFACTOR** method. Algorithm 1 shows a pseudo code for the **RO\_BY\_COFACTOR** method.

---

**Algorithm 1**

---

**Description** Pseudocode for the read-once by cofactor composition.

**Input** A Boolean function  $f(X)$ .

**Output** A read-once tree representing  $f$  or **NULL** if it is not possible.

---

**RO\_BY\_COFACTOR**(Input:  $f(X)$ )

**begin**

- 1: **if** ( $f \equiv \mathbf{0}$ ) **return** 0
- 2: **if** ( $f \equiv \mathbf{1}$ ) **return** 1
- 3:  $x_i := \mathbf{get\_next\_var}(X)$
- 4:  $T_{neg} := \mathbf{RO\_BY\_COFACTOR}(f_{x_i=0})$
- 5:  $T_{pos} := \mathbf{RO\_BY\_COFACTOR}(f_{x_i=1})$
- 6:  $T := \mathbf{RO\_COMPOSITION}(x_i, T_{neg}, T_{pos})$
- 7: **return**  $T$

**end**

---

Since constant functions are read-once by definition, line 1 in Algorithm 1 checks if the target function is equivalent to constant **0**. If it is the case, then a read-once tree with the constant zero node is returned. Line 2 tests if the target function is a constant **1** and returns the appropriate read-once tree if it is the case. In line 3, a next variable is chosen for cofactoring. For the method's point of view, an arbitrary choice can be performed.

The **get\_next\_var** step can take advantage of the structural information present on the data structure used to represent the target function  $f$ . For example, in a BDD representation, the best variable to be cofactored is the variable on the top of the BDD. When the target function is represented by an SOP, then the most binate variable should be selected (BRAYTON ET AL, 1984). When representing  $f$  as a truth-table, the most significant variable should be selected, i.e. the variable with the lowest frequency.

Lines 4 and 5 in Algorithm 1 first obtain a negative and positive cofactor of target variable  $x_i$  and, recursively, find a read-once representation for them. Finally, in line 6, the **RO\_COMPOSITION** method is performed. This method uses only information of both negative and positive read-once trees and decides, whenever possible, which is the insertion point for  $x_i$  in the larger tree. The **RO\_COMPOSITION** method will be explained in the next section.

In the following, an example of the **RO\_BY\_COFACTOR** method presented in Algorithm 1 will be discussed. Let  $f$  be an input function for the **RO\_BY\_COFACTOR** method, represented by a BDD realizing the function  $F = x_1x_2 + x_1x_3$ . Since  $f$  is not constant, the method is called recursively with the negative and positive cofactors w.r.t. variable  $x_i$ . The variable  $x_i$  was first selected because is the variable that is on top of the BDD. The method runs until reaching some constant node, and then recursively returns composing the found read-once trees. Two examples are depicted in Figure 17. A full BDD is used as input example in Figure 17 (left) and a ROBDD is used as example in Figure 17 (right). The read-once expressions found are annotated near to each BDD node. Notice that both BDDs represent the same functions.

Given a BDD with  $m$  nodes and  $n$  inputs, the **RO\_BY\_COFACTOR** method (Algorithm 1) needs to visit each BDD node only once. For each node, one call to **RO\_COMPOSITION** is performed. As will be discussed in the next section, **RO\_COMPOSITION** runs in  $O(n)$ . Consequently, the **RO\_BY\_COFACTOR** has a worst-case performance of  $O(mn)$ .

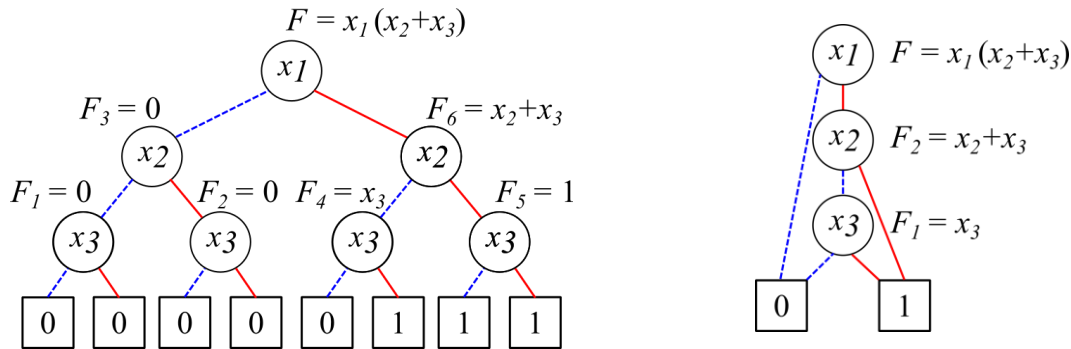


Figure 17 – Two examples of the **RO\_BY\_COFACTOR** algorithm using the input function  $F = x_1x_2 + x_1x_3$ . A full BDD (left) and an ROBDD (right) representing  $f$  are shown.

### 4.2.3 Composing read-once cofactor trees

A cofactor operation over a read-once function results in a read-once function. This property is explored in the **RO\_BY\_COFACTOR** by composing read-once trees from cofactors. The method that composes, whenever possible, read-once cofactor trees into a read-once form is presented in the following.

**Definition 9:** The *arg\_max* function operating on trees  $T_1, T_2$  returns the larger tree regarding support count, i.e.  $T_1$  is returned if  $|\text{sup}(T_1)| \geq |\text{sup}(T_2)|$  or  $T_2$  is returned otherwise. Ties are broken by returning tree  $T_1$ .

**Definition 10:** The lowest common ancestor (LCA) of a set of variable nodes  $\Delta$  in a rooted tree  $T$ ,  $\text{LCA}(T, \Delta)$ , is the lowest (deepest, farthest from the root node) node  $n_i$  such that  $\Delta \subseteq \text{sup}(n_i)$ .

**Definition 11:** An LCA node  $n_i$  of a set of variable nodes  $\Delta$  is considered *complete* if, for each child node  $n_j \in N(n_i)$ , the support of  $n_j$  is either a subset of  $\Delta$  or a disjoint set from  $\Delta$ , i.e.  $\text{sup}(n_j) \subseteq \Delta$  or  $\text{sup}(n_j) \cap \Delta = \emptyset$ .

Let a read-once function  $F = (x_1 x_2 + x_3 + x_4) (x_5 + x_6)$  be represented by the tree  $T$  shown in Figure 18. Node  $n_2$  is a complete LCA for the variable set  $\{x_1, x_2, x_4\}$ , i.e.  $\text{complete\_LCA}(T, \{x_1, x_2, x_4\}) = n_2$ . Node  $n_4$  is the LCA for  $\{x_4, x_5, x_6\}$ , but is not a complete LCA, since  $\text{sup}(n_2) \not\subseteq \{x_4, x_5, x_6\}$  and  $\text{sup}(n_2) \cap \{x_4, x_5, x_6\} \neq \emptyset$ . Indeed, there is not a complete LCA for  $\{x_4, x_5, x_6\}$  for the given tree  $T$ , i.e.  $\text{complete\_LCA}(T, \{x_4, x_5, x_6\}) = \emptyset$ .

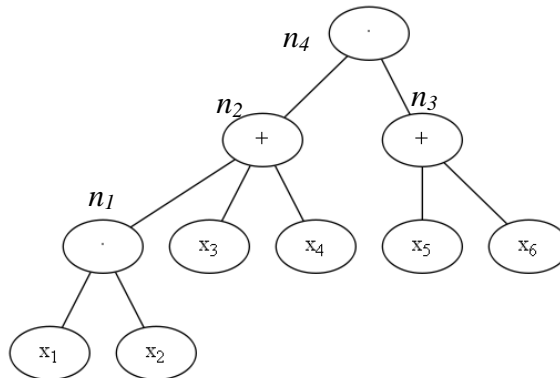


Figure 18 – Lowest common ancestor (LCA) example: The complete LCA for the variable set  $\{x_1, x_2, x_4\}$  is node  $n_2$ . Node  $n_4$  is the LCA for  $\{x_4, x_5, x_6\}$ . For this example, there is not a complete LCA for  $\{x_4, x_5, x_6\}$ .



Let  $f(X)$  be a Boolean function depending on  $n$  inputs, i.e.  $n = |X|$ , and  $x_i \in X$ . The method **RO\_COMPOSITION** receives as input a target variable  $x_i$  and two read-once trees  $T_{neg}$  and  $T_{pos}$ . It is assumed that trees  $T_{neg}$  and  $T_{pos}$  are in canonical form (collapsed and sorted) and represent negative ( $f_{x_i=0}$ ) and positive ( $f_{x_i=1}$ ) cofactors w.r.t. variable  $x_i$ , respectively. Notice that one or both trees could be  $\emptyset$  (**NULL**), meaning that is not possible to represent one or both  $x_i$ 's cofactors as a read-once realization. By definition, if at least one cofactor w.r.t some input variable  $x_i$  is not read-once, the target function is not read-once.

Algorithm 2 shows a pseudo code for the **RO\_COMPOSITION** method. Line 1 checks if at least one of the trees is  $\emptyset$  (**NULL**) and returns  $\emptyset$  if it is the case. Lines 2 and 3 check if both trees represent the same constant function and return the appropriate result. This covers cases 1 and 2 from Table 7. Cases 3 and 4 from Table 7 are covered by lines 4 and 5. Line 6 covers cases 5 and 6, while line 7 covers cases 11 and 12. Cases 7 and 8 are covered by line 8, and line 9 covers cases 9 and 10. In summary, cases 1-12 from Table 7 are covered by lines 1-9 in Algorithm 2.

Line 10 covers the cases where both trees represent the same function, meaning that variable  $x_i$  is redundant so either  $T_{neg}$  or  $T_{pos}$  can be returned. This can be shown as follows. Let  $f$  be represented by the Shannon expansion w.r.t  $x_i$ :  $f = \bar{x}_i \cdot f_{\bar{x}_i} + x_i \cdot f_{x_i}$ . If the trees match, i.e. represent the same function, then  $f_{\bar{x}_i} \equiv f_{x_i} \equiv \hat{f}$ . By substitution,  $f = \bar{x}_i \cdot \hat{f} + x_i \cdot \hat{f}$  which is equivalent to  $f = \hat{f}(\bar{x}_i + x_i) \equiv \hat{f}(1) \equiv \hat{f}$ . Cases 13-20 are covered by lines 11-20 and require a more detailed explanation.

**Lemma 1:** Let  $f(X)$  be an  $n$ -input function depending on two or more inputs, i.e.  $n > 1$ . Let  $x_i$  represent an arbitrary variable in  $X$ , and let  $n_0$  ( $n_1$ ) denote the number of variables that the negative (positive) cofactor of  $x_i$  in  $f$  depends on. If  $f$  is a read-once function, then  $n_0 = (n - 1)$  with  $n_1 < n_0$  or  $n_1 = (n - 1)$  with  $n_0 < n_1$ .

**Proof:** Let  $T$  be a read-once tree representing an  $n$ -input read-once function  $f$ . Since  $n > 1$ , the parent node of  $x_i$  in  $T$  must be an operator node. Let  $n_j$  be this operator node. When  $x_i$  is cofactored, the assigned value is propagated over the tree  $T$ , reaching node  $n_j$ . If  $n_j$  is an AND (OR) operator, a 0 (1) value will be propagated to  $n_j$ 's parent node making all children nodes from  $n_j$  redundant (don't care). After the propagation operation, it is easy to see that  $x_i$  and variables in the support of its sibling's nodes will be redundant. Conversely, when the value reaching  $n_j$  is not a dominating, i.e. value 1 (0) for an AND (OR) operator, then the AND

(OR) of all children from  $n_j$  besides  $x_i$  will be propagated to  $n_j$ 's parent node. This is the case where just  $x_i$  will be redundant after the cofactor operation. ■

**Example:** Let  $T = x_1(x_2 + x_3(x_4 + x_5))$  represent a 5-input read-once function  $f$ . Let  $T_{neg}$  and  $T_{pos}$  represent negative and positive cofactors w.r.t.  $x_3$  in  $f$ , such that  $T_{neg} = x_1x_2$  and  $T_{pos} = x_1(x_2 + x_4 + x_5)$ . Let  $n_0 = |\text{sup}(T_{neg})| = 2$  and  $n_1 = |\text{sup}(T_{pos})| = 4$ . Since  $n = 5$ ,  $n_1 = (n - 1)$  and  $n_0 < n_1$ , Lemma 1 holds. More examples are shown in cases 5-20 in Table 7. Notice that Lemma 1 is a necessary but not sufficient condition for identifying read-once functions.

**Lemma 2:** Let  $f(X)$  be a read-once function,  $x_i \in X$ ,  $T_{neg}$  and  $T_{pos}$  represent read-once trees realizing  $f_{x_i=0}$  and  $f_{x_i=1}$ , respectively. According to Lemma 1, one of the cofactors must have more inputs in the support. Let  $T_{larger}$  be the larger (in support count) tree and  $T_{smaller}$  be the other tree. Then the support set of the smaller tree should be a proper subset of the support set of the larger tree, i.e.  $\text{sup}(T_{smaller}) \subset \text{sup}(T_{larger})$ .

**Proof:** Straightforward from Lemma 1 and its proof. ■

Lines 11 – 13 of Algorithm 2 store in  $T_{larger}$  and  $T_{smaller}$  the larger and smaller cofactor trees, respectively, and check if Lemma 1 Lemma 2 holds. If it is not the case, then the function is not read-once and  $\emptyset$  (**NULL**) is returned.

**Theorem 3:** Let  $f(X)$  be a non-trivial read-once function, i.e. the both cofactors regarding  $x_i \in X$  are not constant functions. Let  $T_{neg}$  and  $T_{pos}$  represent read-once trees realizing  $f_{x_i=0}$  and  $f_{x_i=1}$ , respectively. If  $f$  is a read-once function, then according to Lemma 1, one cofactor tree is larger than the other regarding support count. Let  $T_{larger}$  be the larger tree,  $T_{smaller}$  be the smaller tree and  $\Delta$  represent the missing variables in  $T_{smaller}$  but present in  $T_{larger}$ , i.e.  $\Delta := \text{sup}(T_{larger}) \setminus \text{sup}(T_{smaller})$ . If  $f(X)$  is a non-trivial read-once function, a complete lowest common ancestor (CLCA) node  $n_{CLCA}$  from  $T_{larger}$  for the set of variables  $\Delta$  must exist, i.e. ( $n_{CLCA} \neq \emptyset$ ).

**Proof (by contradiction):** Let us assume that  $f$  is a non-trivial read-once and no complete-LCA node is found for  $\Delta$ . However, by definition at least one non-complete LCA node  $n_i$  must exist in  $T_{larger}$ . Since node  $n_i$  is a LCA but is not complete, there must exist at least one child node of  $n_i$ ,  $n_j \in N(n_i)$ , such that or 1)  $\text{sup}(n_j) \not\subseteq \Delta$  **and**  $\text{sup}(n_j) \cap \Delta \neq \emptyset$  or 2)  $\text{sup}(n_j) \subseteq \Delta$  **and**  $\text{sup}(n_j) \cap \Delta = \emptyset$  holds. By simplification, only  $\text{sup}(n_j) \cap \Delta \neq \emptyset$  rule

remains in case 1. Let  $\aleph = \text{sup}(n_j) \cap \Delta$ . This means that by cofactoring  $x_i$ , some variables in  $\aleph$  are missing and some variables are present in  $T_{smaller}$ . However, since  $f$  is read-once, after cofactoring  $x_i$  each group of variables that is missing must belong to the same sub-tree which becomes redundant after cofactoring. This contradicts case 1 since some variables are missing and some are not. Case 2 is a contradiction since the only case where both conditions evaluate to true is the case where both  $\text{sup}(n_j)$  and  $\Delta$  are empty, i.e.  $\text{sup}(n_j) = \Delta = \emptyset$ , which contradicts the fact that at least one element must be in  $\Delta$ , i.e. the tree  $T_{larger}$  must have at least one variable more than  $T_{smaller}$ . ■

After checking if both Lemma 1 and Lemma 2 holds, the algorithm stores in  $\Delta$  the set of variables that are missing in  $T_{smaller}$  but are present in  $T_{larger}$  and obtain  $n_{CLCA}$ , a complete lowest common ancestor (CLCA) node from  $T_{larger}$  for the set of variables  $\Delta$ . If there is not a CLCA node for the set  $\Delta$ , Theorem 3 does not hold and no read-once realization can be obtained for the given input and  $\emptyset$  (**NULL**) is returned.

**Definition 12:** Let  $T$  represent a read-once tree and  $\Delta$  be a set of variables. The removal of variables  $\Delta$  in  $T$ , denoted  $T \setminus \Delta$ , consists of two steps: 1) leaf nodes contained in  $\Delta$  are deleted from  $T$  and 2) the resulting tree is canonized (collapsed and sorted).

Figure 19 depicts a step by step example of the removal operation. Let the initial read-once tree be represented by  $T = x_2 + x_3 + x_4x_5 + x_6x_7$ , as shown in (a), and  $\Delta = \{x_2, x_3, x_6, x_7\}$ . Nodes are first deleted (b)-(d) and the resultant tree is collapsed and ordered (e)-(g). The resulting tree is canonical.

**Definition 13:** A *non-controlling value* related to an input  $x_i$  of an AND (OR) operation is an input assignment such that a value 1 (0) is propagated to an AND (OR) input.

**Lemma 3:** Let  $n_i$  be a child node of an operator node  $n_j$  in a read-once tree, i.e.  $n_i \in N(n_j)$ . Deleting a node  $n_i$  from  $n_j$ , i.e.  $n_j \setminus n_i$ , is equivalent to assigning a value  $\alpha_i$  to node  $n_i$  such that a non-controlling value is propagated to the input of node  $n_j$ .

---

**Algorithm 2**


---

**Description** Pseudocode for the read-once composition method.

**Input** An input variable  $x_i$  and two **canonized** trees  $T_{neg}$  and  $T_{pos}$  representing negative and positive cofactors w.r.t input  $x_i$ , respectively.

**Output** A read-once tree by inserting  $x_i$  in the larger tree or  $\emptyset$  (**NULL**) if it is not possible.

---

**RO\_COMPOSITION**(Input:  $x_i, T_{neg}, T_{pos}$ )

**begin**

```

1: if ( $T_{neg} = \emptyset \vee T_{pos} = \emptyset$ ) return  $\emptyset$ 
2: if ( $T_{neg} = 0 \wedge T_{pos} = 0$ ) return 0
3: if ( $T_{neg} = 1 \wedge T_{pos} = 1$ ) return 1
4: if ( $T_{neg} = 0 \wedge T_{pos} = 1$ ) return  $x_i$ 
5: if ( $T_{neg} = 1 \wedge T_{pos} = 0$ ) return  $\bar{x}_i$ 
6: if ( $T_{neg} = 0$ ) return CANONIZE( $x_i \cdot T_{pos}$ )
7: if ( $T_{neg} = 1$ ) return CANONIZE( $\bar{x}_i + T_{pos}$ )
8: if ( $T_{pos} = 0$ ) return CANONIZE( $\bar{x}_i \cdot T_{neg}$ )
9: if ( $T_{pos} = 1$ ) return CANONIZE( $x_i + T_{neg}$ )
10: if MATCH( $T_{neg}, T_{pos}$ ) return  $T_{neg}$ 
11:  $T_{larger} := \mathbf{ARG\_MAX}(T_{neg}, T_{pos})$ 
12:  $T_{smaller} := \begin{cases} T_{neg}, & \text{if } T_{larger} = T_{pos} \\ T_{pos}, & \text{otherwise} \end{cases}$ 
13: if ( $\text{sup}(T_{smaller}) \not\subseteq \text{sup}(T_{larger})$ ) return  $\emptyset$ 
14:  $\Delta := \text{sup}(T_{larger}) \setminus \text{sup}(T_{smaller})$ 
15:  $n_{CLCA} := \mathbf{COMPLETE\_LCA}(T_{larger}, \Delta)$ 
16: if ( $n_{CLCA} = \emptyset$ ) return  $\emptyset$ 
17:  $T_{modified} := \mathbf{CANONIZE}(T_{larger} \setminus \Delta)$ 
18: if not MATCH( $T_{modified}, T_{smaller}$ ) return  $\emptyset$ 
19:  $T := \mathbf{COMPOSE}(x_i, T_{larger}, n_{CLCA}, \Delta)$ 
20: return  $T$ 

```

**end**

---

Let the read-once tree  $T$  in Figure 19 (a) represent a function  $f$ . Node  $x_2$  was removed from  $T$  in (b), leading to a new read-once tree  $T_1$ . The removal operation is equivalent to propagating a non-controlling value  $\mathbf{0}$  to the input of the OR operator, i.e.  $f_{\alpha_2} = f_1$  with  $\alpha_2 = \{x_2 = 1\}$ . The same process is repeated in (c). Let tree  $T_3$  be equivalent to the tree shown in (c) and represent function  $f_3$ . The removal of the node  $x_6$  from tree  $T_3$  result in tree  $T_4$  (d) and is equivalent to propagating a non-controlling value  $\mathbf{1}$  to the input of the AND (right) operator, i.e.  $f_{3_{\alpha_6}} = f_4$  with  $\alpha_6 = \{x_6 = 1\}$ .

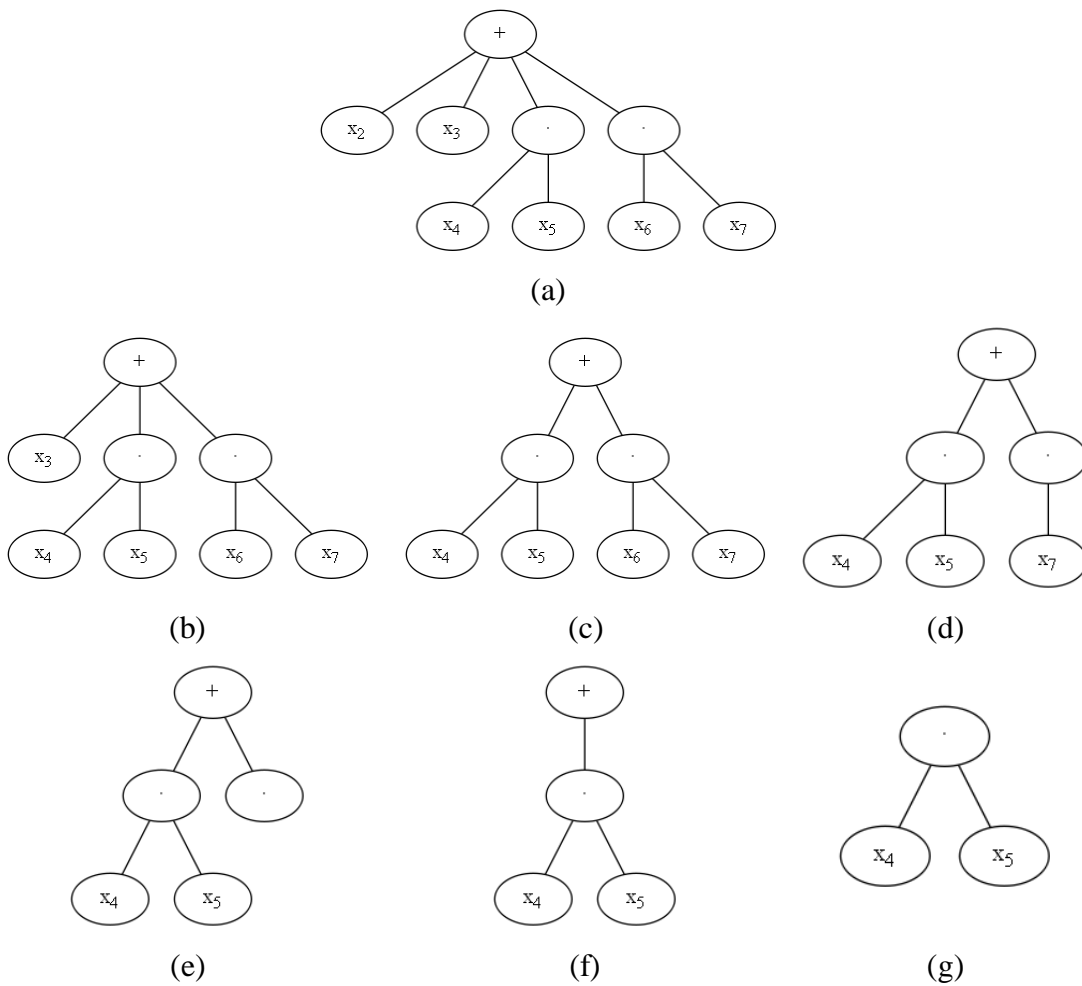


Figure 19 – Step by step example of removing variables  $\Delta = \{x_2, x_3, x_6, x_7\}$  from a read-once tree  $T = x_2 + x_3 + x_4x_5 + x_6x_7$ . (a) Original tree. (b) Node  $x_2$  removed. (c) Node  $x_3$  removed. (d) Node  $x_6$  removed. (e) Node  $x_7$  removed. (f) Operator node “·” simplified. (g) Operator node “+” simplified.

**Lemma 4:** Let  $T_{neg}$  and  $T_{pos}$  represent two non-constant read-once trees realizing  $f_{x_i=0}$  and  $f_{x_i=1}$ , respectively. Let  $T_{larger}$  be the larger tree,  $T_{smaller}$  be the smaller tree and  $\Delta$

represent the missing variables in  $T_{smaller}$  that are present in  $T_{larger}$ . Let  $T_{modified}$  represent the tree after the removal operation of  $\Delta$  from  $T_{larger}$ , i.e.  $T_{modified} := T_{larger} \setminus \Delta$ . Let  $\alpha$  be a cube-cofactor representing the assignment of variables applied to variable set  $\Delta$  during the removal from  $T_{larger}$ . If  $f$  is a read-once function,  $T_{modified}$  must match  $T_{smaller}$ .

**Proof:** Straightforward from Lemma 3 and Theorem 1. ■

**Corollary:** If  $T_{modified}$  and  $T_{smaller}$  match, then  $f_{x_i=1,\alpha} \equiv f_{x_i=0}$  if  $T_{larger} = T_{pos}$ . If  $T_{larger} = T_{neg}$ , then  $f_{x_i=0,\alpha} \equiv f_{x_i=1}$ .

Line 17 from Algorithm 2 shows the process of obtaining a modified tree  $T_{modified}$ , obtained after the removal of  $\Delta$  from  $T_{larger}$ . In line 18, the algorithm tests if Lemma 4 holds, and returns **NULL** if it is not the case. If it is the case that  $T_{modified}$  matches  $T_{smaller}$ , we can guarantee that there exists a read-once realization for the target function. Line 19 then performs the **COMPOSE** method, which based the CLCA node  $n_{CLCA}$ , the variable set  $\Delta$ , and the information if the  $T_{larger}$  is the negative or positive cofactor, modify  $n_{CLCA}$  and insert the target variable  $x_i$ .

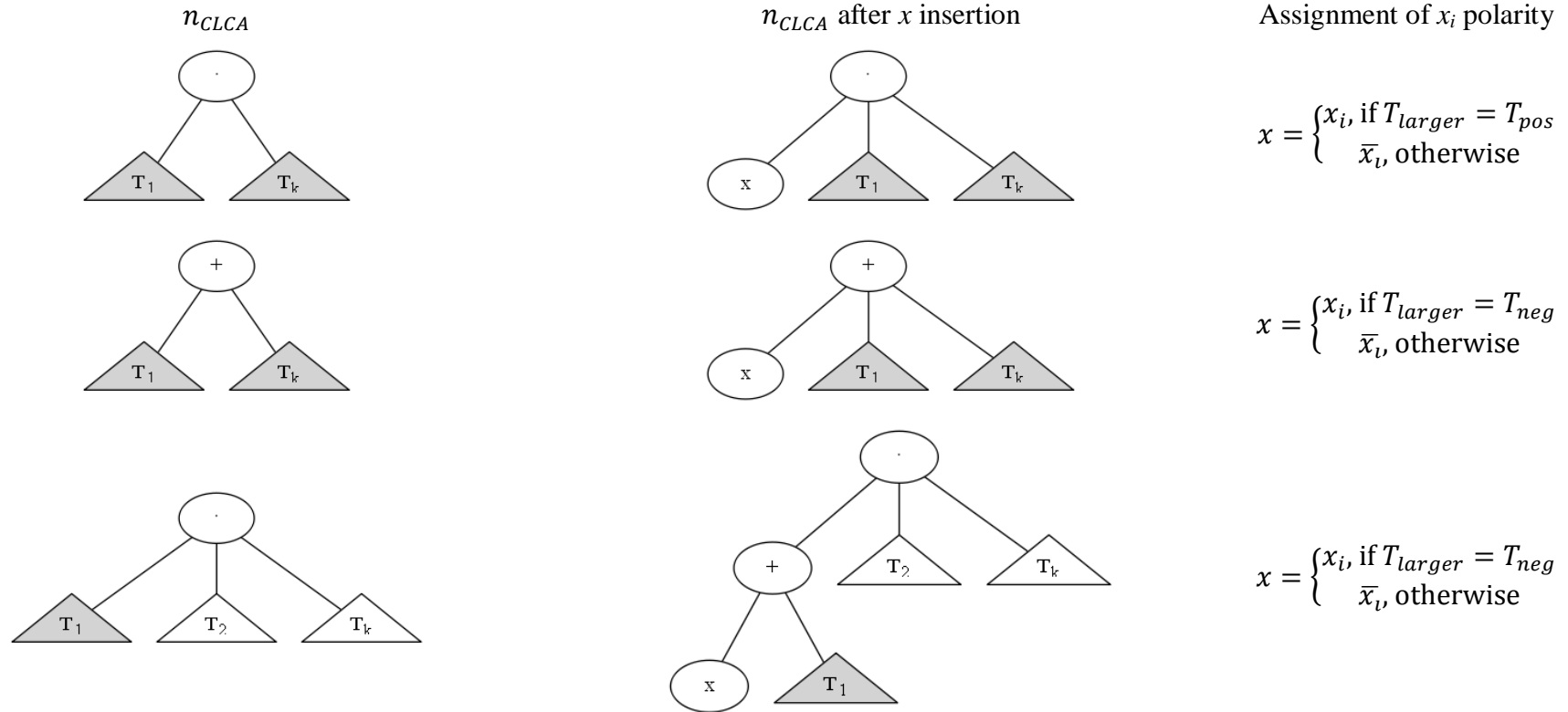
Table 8 presents all possible rules for composition of the target variable  $x_i$  by modifying node  $n_{CLCA}$ . The first column shows the original CLCA nodes. The second column depicts the state of the tree after the insertion of the target variable  $x_i$ . The last column presents the test in order to properly assign the polarity of the inserted literal. Nodes depicted by triangles denote read-once subtrees. Filled triangles represent those subtrees  $T_i$  where all support variables are missing, i.e.  $\text{sup}(T_i) \subseteq \Delta$ . First two rows show the case when all children of the  $n_{CLCA}$  are missing. This is the case when the variable must be inserted directly as a new child of  $n_{CLCA}$ .

The third and fourth rows of Table 8 show the cases where exactly one child of  $n_{CLCA}$ , say  $T_1$ , has support variables missing. In this case, a new operator node  $n_j$ , with functionality opposed from the  $n_{CLCA}$  node, must be created. Subtree  $T_1$  is removed from  $n_{CLCA}$  and added as a child of node  $n_j$  together with the target variable  $x_i$ . Finally node  $n_j$  is added to  $n_{CLCA}$ .

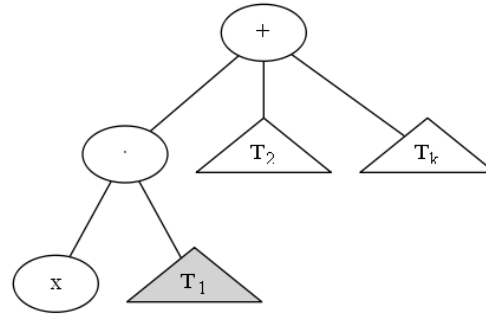
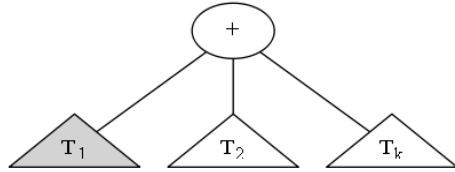
The last two rows presented in Table 8 show cases when at least two subtrees, say  $T_1, T_2$ , are missing but some other subtrees are not. Then a new operator node  $n_k$ , with the same functionality of the  $n_{CLCA}$  node is created. Subtrees  $T_1, T_2$  are removed from  $n_{CLCA}$  and added as child of node  $n_k$ . Then a new operator node  $n_j$ , with functionality opposed from the  $n_{CLCA}$  node, must be created. Both nodes  $n_k$  and the target variable are added as child nodes of  $n_j$ . Finally node  $n_j$  is added to  $n_{CLCA}$ .

Let  $n$  be the number of inputs of the larger tree used as input of the **RO\_COMPOSE** method. In composing two read-once trees with at most  $n$  inputs, **RO\_COMPOSE** (Algorithm 2) has a worst-case performance of  $O(n)$ . Testing where input trees are constants takes  $O(1)$ . The **MATCH** method compares the structure of each tree such that each node in the tree is visited once, which takes in the worst-case  $O(n)$ . The **CANONIZE** method first collapses the input tree which takes  $O(n)$  and then the resulting tree is sorted, which takes  $O(n \log n)$ . However, the **RO\_COMPOSE** method assumes that the input trees are already in canonical form. This is a fair assumption since the method is bottom-up and canonizes trees by composition. In this sense, since trees are already sorted, **CANONIZE** can run in linear time w.r.t the number of inputs  $O(n)$ . This can be achieved by propagating the information of the new inserted node, and updating the parent node's children ordering. Finding the complete LCA node runs in  $O(n)$ . The last procedure **RO\_COMPOSE** also runs in  $O(n)$ .

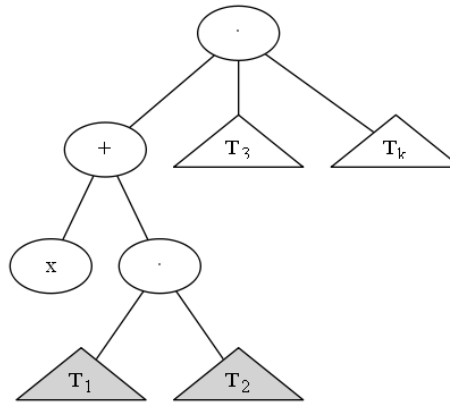
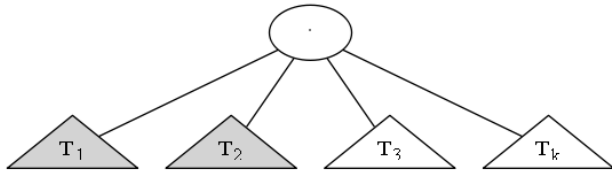
Table 8 – Enumeration of all possible read-once trees considering the complete lower common ancestor node  $n_{CLCA}$ . Nodes depicted by triangles denote read-once subtrees. Filled triangles represent those subtrees  $T_i$  where all support variables are missing, i.e.  $\text{sup}(T_i) \subseteq \Delta$ .



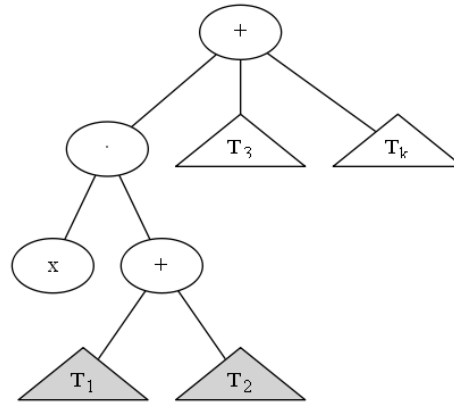
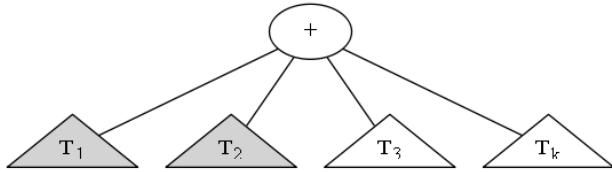




$$x = \begin{cases} x_i, & \text{if } T_{larger} = T_{pos} \\ \bar{x}_i, & \text{otherwise} \end{cases}$$



$$x = \begin{cases} x_i, & \text{if } T_{larger} = T_{neg} \\ \bar{x}_i, & \text{otherwise} \end{cases}$$



$$x = \begin{cases} x_i, & \text{if } T_{larger} = T_{pos} \\ \bar{x}_i, & \text{otherwise} \end{cases}$$

### 4.3 Results

We implemented the proposed method using ROBDDs as a basic data structure. Notice that the method does not depend on a specific type of data structure. However, BDDs are a natural candidate since cofactors are conveniently already evaluated. The platform used to perform these results was a Linux system with Intel Core i5 2400 processor and 4GB main memory.

The first experiment consists of the isolated analysis of the **RO\_COMPOSITION** method. The method receives two canonical read-once trees and, based on this information, composes a new read-once tree for the target function. Figure 20 shows that **RO\_COMPOSITION** runs linearly with the number of inputs.

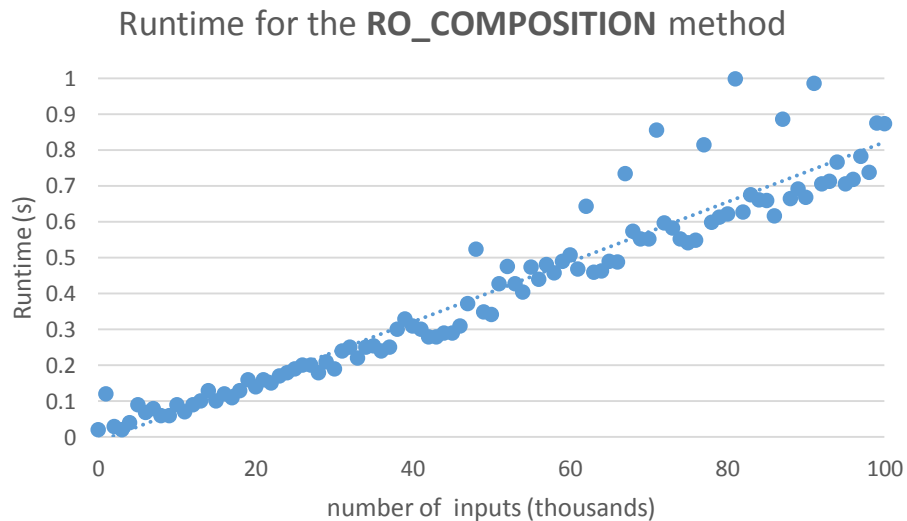


Figure 20 – Runtime analysis of the **RO\_COMPOSITION** method. The number of inputs are in thousands. Results support the claim that the **RO\_COMPOSITION** runs in  $O(n)$ .

The second experiment consists of obtaining ROBDDs for read-once functions. The variable order chosen to create the ROBDDs was completely random. Notice the generated ROBDDs could result in fewer nodes by using a dynamic variable ordering (RUDELL, 1993). However, our idea is to show how the method performs even when the number of BDD nodes is very large. Results by running **RO\_BY\_COFACTOR** on these BDDs are shown in Table 9. Since we used a random variable order, the largest BDD with 637,185 nodes was the one representing a read-once function with 51 inputs and took 8.76 seconds to complete. In all

cases, read-once solutions were found for each BDD node. In total, 1,709,418 BDD nodes were evaluated, taking 20.47 seconds to complete.

In the third experiment, we tested the method over non-read-once functions. Since we know in advance that the BDD does not represent a read-once function, no read-once solution should be found for the top node. However, there must be some read-once realization for some BDD nodes, at least for the nodes representing constants and input variables.

Let a random, non-read-once function be represented by  $F_1 = \bar{x}_1 x_3 \bar{x}_4 x_5 x_6 + \bar{x}_2 x_3 x_4 \bar{x}_5 x_6 + x_1 \bar{x}_2 x_5 x_6 + x_2 \bar{x}_3 \bar{x}_5 \bar{x}_6 + x_1 x_3 x_5 x_6 + \bar{x}_1 x_2 x_3 x_4 x_5 + \bar{x}_1 \bar{x}_2 \bar{x}_4 x_6 + \bar{x}_1 \bar{x}_2 x_3 x_4 \bar{x}_5 + x_1 \bar{x}_3 \bar{x}_5 \bar{x}_6 + x_1 x_2 \bar{x}_3 \bar{x}_5 + x_2 x_3 \bar{x}_4 \bar{x}_5 x_6 + \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 + x_1 x_2 \bar{x}_4 x_5 \bar{x}_6 + x_1 x_2 x_4 \bar{x}_5 \bar{x}_6 + x_1 \bar{x}_3 x_4 x_5 + \bar{x}_1 x_2 x_3 \bar{x}_4 \bar{x}_6$ . The resulting BDD is shown in Figure 21. Nodes in green represent read-once functions.

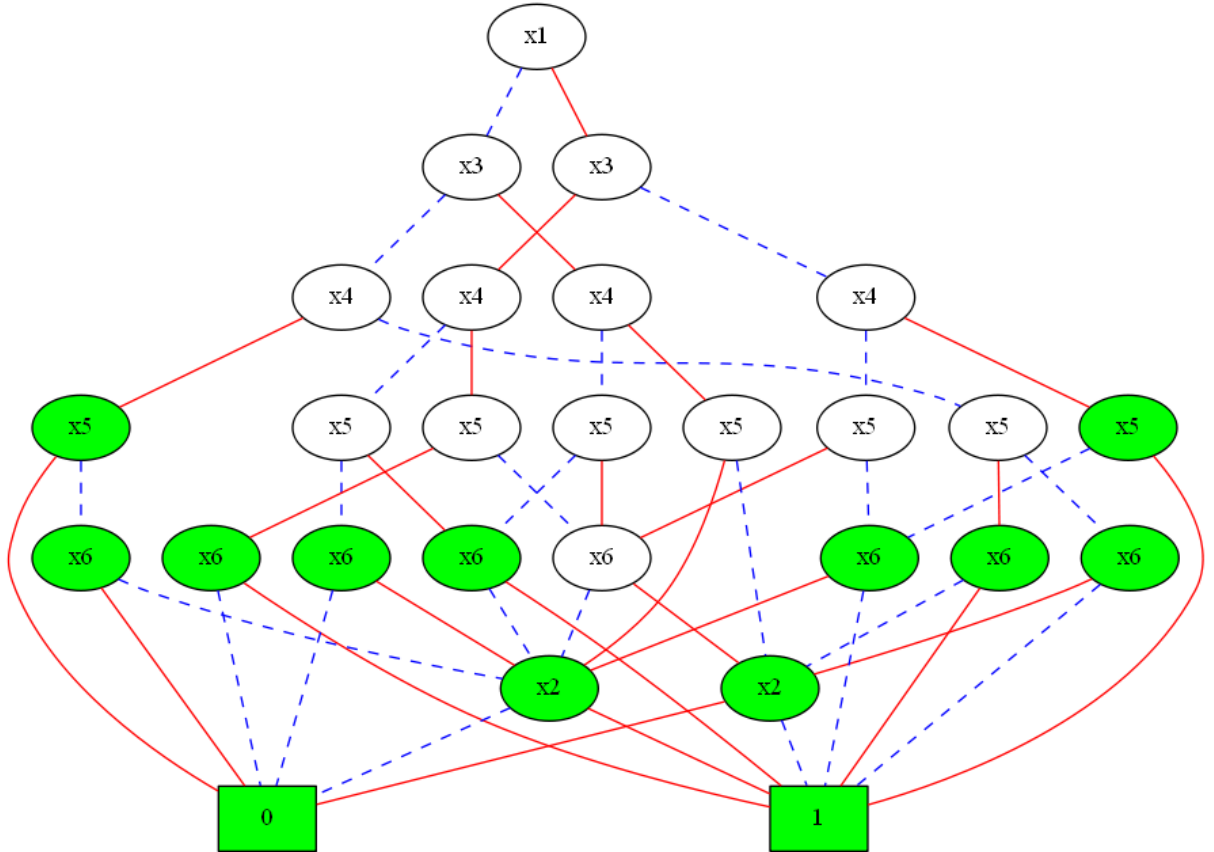


Figure 21 – ROBDD representing a non-read-once function  $F_1 = \bar{x}_1 x_3 \bar{x}_4 x_5 x_6 + \bar{x}_2 x_3 x_4 \bar{x}_5 x_6 + x_1 \bar{x}_2 x_5 x_6 + x_2 \bar{x}_3 \bar{x}_5 \bar{x}_6 + x_1 x_3 x_5 x_6 + \bar{x}_1 x_2 x_3 x_4 x_5 + \bar{x}_1 \bar{x}_2 \bar{x}_4 x_6 + \bar{x}_1 \bar{x}_2 x_3 x_4 \bar{x}_5 + x_1 \bar{x}_3 \bar{x}_5 \bar{x}_6 + x_1 x_2 \bar{x}_3 \bar{x}_5 + x_2 x_3 \bar{x}_4 \bar{x}_5 x_6 + \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 + x_1 x_2 \bar{x}_4 x_5 \bar{x}_6 + x_1 x_2 x_4 \bar{x}_5 \bar{x}_6 + x_1 \bar{x}_3 x_4 x_5 + \bar{x}_1 x_2 x_3 \bar{x}_4 \bar{x}_6$ .

Nodes in green represent read-once functions.

Table 9 – Runtime results after running **RO\_BY\_COFACTOR** over ROBDDs representing read-once functions. The ROBDDs were constructed using a random variable order.

Inputs	BDD nodes	Runtime (sec)	Inputs	BDD nodes	Runtime (sec)
2	4	0.02	27	1,146	0.01
3	5	0.00	28	2,402	0.05
4	8	0.00	29	3,808	0.05
5	9	0.00	30	3,604	0.04
6	10	0.00	31	3,948	0.04
7	11	0.00	32	4,104	0.06
8	20	0.00	33	7,882	0.10
9	28	0.00	34	1,133	0.01
10	34	0.00	35	7,052	0.06
11	44	0.00	36	3,614	0.03
12	42	0.00	37	18,674	0.15
13	68	0.00	38	31,516	0.26
14	51	0.00	39	14,344	0.12
15	114	0.00	40	5,794	0.04
16	80	0.01	41	21,920	0.23
17	92	0.00	42	33,238	0.28
18	353	0.01	43	22,868	0.16
19	368	0.01	44	130,210	1.26
20	718	0.02	45	61,361	0.61
21	412	0.00	46	152,138	1.79
22	389	0.01	47	134,976	1.49
23	685	0.01	48	50,636	0.54
24	1,079	0.01	49	284,700	3.62
25	851	0.01	50	65,116	0.61
26	574	0.00	51	637,185	8.76

Total BDD nodes: 1,709,418

Total Runtime: 20.47 sec

## 4.4 Conclusions

A synthesis method that finds, whenever it is possible, a read-once realization for a target function was proposed. The method was designed based on a divide-and-conquer strategy. Finding a read-once tree for a target function consists of obtaining read-once trees for simpler sub-problems: negative and positive cofactors. These solutions are then composed (conquer phase), resulting in a read-once solution for the original problem (target function). The method is independent of the Boolean function's data structure representation. It relies only on cofactor operation and equivalence checking regarding constants.

Given a BDD with  $m$  nodes and  $n$  inputs, the **RO\_BY\_COFACTOR** method (Algorithm 1) needs to visit each BDD node only once. For each node, one call to **RO\_COMPOSITION** is performed. As discussed in the section 4.2.3, **RO\_COMPOSITION** runs in  $O(n)$ . Consequently, the **RO\_BY\_COFACTOR** has a worst-case performance of  $O(mn)$ .

As future work, we will investigate how to modify this presented method to handle not only read-once but also disjoint-support decomposable functions.

## 5 DISJOINT-SUPPORT DECOMPOSABLE FUNCTIONS

The representation of complex Boolean functions through simpler subfunctions is one of the main tasks comprising the logic synthesis process. In general, both the number of gates and the execution time tend to increase exponentially with the number of inputs of the target function.

A Boolean function  $F(X)$  can be expressed through subfunctions  $G$  and  $H$ , such that:

$$f(X) = h(g(X_1), X_2) \quad (12)$$

where  $X_1$  and  $X_2 \neq \emptyset$ , and  $X_1 \cup X_2 = X$ . If such a representation exists, it is considered a functional decomposition of  $f$ , where  $g$  and  $h$  are called composition and decomposition functions, respectively. A simple disjoint-support decomposition (DSD) is a special case of functional decomposition, where the input sets  $X_1$  and  $X_2$  do not share any element, *i.e.*,  $X_1 \cap X_2 = \emptyset$  (ASHENHURST, 1957), (CURTIS, 1962). The interest in these functions is due to low implementation cost, since optimal DSD implementations grow linearly with the number of inputs. DSD functions have been applied to different IC design domains including ASIC design, FPGA design, and digital circuit verification.

This chapter presents two approaches for synthesis of DSD functions. A top-down approach checks if there is an OR, AND, or XOR decomposition based on sum-of-products (SOP), product-of-sums (POS) and exclusive-sum-of-products (ESOP) inputs, respectively (CALLEGARO ET AL, 2015). This method is presented in Section 5.2. The second method runs in a bottom-up fashion and is based on Boolean difference and cofactor analysis (CALLEGARO ET AL, 2015b). Two simple tests provide sufficient and necessary conditions to identify AND and exclusive-OR (XOR) decompositions. This approach is presented in Section 5.3.

## 5.1 Previous work

Ashenhurst (ASHENHURST, 1957) was one of the first authors to point out the importance of the functional decomposition process. His method was based on decomposition charts, which is not practical for large circuit design. In order to reduce complexity, in (ROTH; KARP, 1962), Roth and Karp introduced a method to represent decomposition charts through a set of cubes. However, the problem was shifted to finding a good variable partitioning in an efficient way. A survey on functional decomposition methods proposed up until 1995 is presented in (PERKOWSKI, 1995).

In (BERTACCO; DAMIANI, 1997), Bertacco and Damiani proposed an approach based on binary decision diagrams (BDD) that performs DSD by traversing the BDD without the aid of a decomposition chart. Matsunaga, in (MATSUNAGA, 2002), presented a new method based on (BERTACCO; DAMIANI, 1997) that performs variable splitting step in a modified way. In (PLAZA; BERTACCO, 2005), Plaza and Bertacco created a framework called *Staccato* that performs DSD based on BDDs and symbolic kernel manipulation.

In (MINATO; DE MICHELI, 1998), Minato and De Micheli proposed an algorithm based on irredundant sum-of-products (ISOP) and factorization. Their algorithm requires a Minato-Morreale ISOP, which means that distinct heuristic methods to improve the ISOP runtime generation like *MINI* (HONG; CAIN; OSTAPKO, 1974), *ESPRESSO* (BRAYTON ET AL, 1984), *Presto-II* (BARTHOLOMEUS; MAN, 1985), *PALMINI* (NGUYEN; PERKOWSKI; GOLDSTEIN, 1987), *ESPRESSO-SIGNATURE* (MCGEER; SANGHAVI; BRAYTON; VINCENTELLI, 1993), *Scherzo* (COUDERT, 1994), and *BOOM* (HLAVICKA; FISER, 2001) could not be used.

Recently, in (MISHCHENKO; BRAYTON, 2013) and (MISHCHENKO, 2014), Mishchenko created a DSD-based framework to perform LUT structure mapping more efficiently. The Mishchenko's decomposition considers 2-to-1 multiplexer (MUX) as a basic type of decomposition.

## 5.2 Top-down decomposition based on SOP, POS and ESOP forms

This section presents an algorithm for synthesis of simple disjoint-support decompositions (CALLEGARO ET AL, 2015). It is based on a variable intersection graph which is directly obtained from a set of Boolean terms representing a Boolean function  $f$  (GOLUMBIC, 2004),



(MINTZ, GOLUMBIC, 2005). If such a graph is disconnected into  $m$  connected components, the function  $f$  can be decomposed into  $m$  subfunctions  $h_0(X_0), \dots, h_{m-1}(X_{m-1})$ .

A decomposition operator  $\circ \in \{\mathbf{AND}, \mathbf{OR}, \mathbf{XOR}\}$  is determined such that the DSD implementation of  $f$  is obtained by  $f = \circ (h_0(X_0), \dots, h_{m-1}(X_{m-1}))$ , where  $X_0, \dots, X_{m-1}$  are mutually disjoint. The decomposition operator  $\circ$  is an **OR** operator if the Boolean terms were obtained from cubes of an ISOP description of  $f$ . The decomposition operator  $\circ$  is an **AND** operator if the Boolean terms were sums from an irredundant product-of-sums (IPOS) description. And finally  $\circ$  is an **XOR** operator when the Boolean terms come from products of an exclusive sum-of-products (ESOP) description of  $f$ .

From our experiments, it has successfully synthesized all possible DSD functions when tested over the set of all DSD functions up to 6 inputs as well as over a selection of PLA benchmarks.

### 5.2.1 Definitions and notation

Let  $F$  be a Boolean formula in SOP, POS or ESOP form representing a function  $f(X)$ .

**Definition 14:** A variable intersection graph (VIG)  $G_F = (X, E)$  is an undirected graph where vertices correspond to variables of  $F$ , and there is an edge between  $(x_i, x_j) \in E$ ,  $x_i, x_j \in X$  if and only if  $x_i$  and  $x_j$  are present in the same Boolean term (GOLUMBIC, 2004), (MINTZ, GOLUMBIC, 2005).

Let  $f = (a+b) \oplus (c \cdot d)$  be represented by the ISOP form  $F = (!a!bcd + a!d + b!c + a!c + b!d)$  and by the ESOP form  $H = 1 \oplus (!a!b) \oplus (c \cdot d)$ . The VIG  $G_F$  and  $G_H$  are shown in Figure 22 (a) and Figure 22 (b), respectively.



Figure 22 – A variable intersection graph obtained from an (a) ISOP

$F = (!a!b \cdot c \cdot d + a \cdot !d + b \cdot !c + a \cdot !c + b \cdot !d)$  and (b) from ESOP form  $H = 1 \oplus (!a!b) \oplus (c \cdot d)$ .

In graph theory, a connected component of an undirected graph is a subgraph in which 1) any two vertices are connected to each other by paths, and 2) is connected to no additional vertices in the supergraph (HOPCROFT, 1973). For example, the graph  $G_F$  presented in Fig.

2(a) is connected, *i.e.* contains exactly one connected component  $\{a, b, c, d\}$ . The graph  $G_H$ , shown in Fig. 2(b), is disconnected and contains two connected components  $\{a, b\}$  and  $\{c, d\}$ .

The problem of finding connected components in undirected graphs is well-known in graph theory and was efficiently solved first by Hopcroft and Tarjan (HOPCROFT, 1973). Currently, the union–find algorithm that uses a disjoint-set data structure is the most efficient way to find connected components (CORMEN ET AL, 2001).

### 5.2.2 Proposed method

Consider a Boolean function  $f(X)$  decomposed into two subfunctions  $f_1(X_1)$  and  $f_2(X_2)$  such that

$$f(X) = f_1(X_1) \circ f_2(X_2) \quad (13)$$

where  $X_1 \cup X_2 = X$ ,  $X_1 \cap X_2 = \emptyset$  and  $\circ \in \{ \cdot, +, \oplus \}$ .

**Theorem 4:** If  $f$  is decomposed as in Eq. (13) and such decomposition is realized by an OR operator, *i.e.* ( $\circ = +$ ), an ISOP representing  $f$  (denoted  $ISOP_f$ ) can be determined as:

$$ISOP_f = ISOP_{f_1} + ISOP_{f_2} \quad (14)$$

**Proof:**  $ISOP_{f_1}$  (by itself) cannot have any literal or cube removed; otherwise it would not be an ISOP. The same can be applied to  $ISOP_{f_2}$ . As  $f_1$  and  $f_2$  have disjoint supports, the cubes from  $f_1$  cannot cover cubes from  $f_2$  and vice versa. Thus, the sum (OR) of the ISOPs of  $f_1$  and  $f_2$  is still an ISOP of  $f$ . ■

**Theorem 5:** If an ISOP representing  $f$  was obtained as in Eq. (13), the resulting VIG from  $ISOP_f$  is disconnected into two connected components  $X_1$  and  $X_2$ .

**Proof:** Since the cubes of  $ISOP_{f_1}$  contain no input variables in  $X_2$  and vice-versa, there is no edge connecting any element from  $X_1$  to  $X_2$ , resulting in a graph partition of two connected components  $X_1$  and  $X_2$ . ■

Figure 23 depicts an example that summarizes Theorem 4 and Theorem 5. A generic decomposition tree and its corresponding VIG are presented in Figure 23 (a) and Figure 23 (b), respectively.

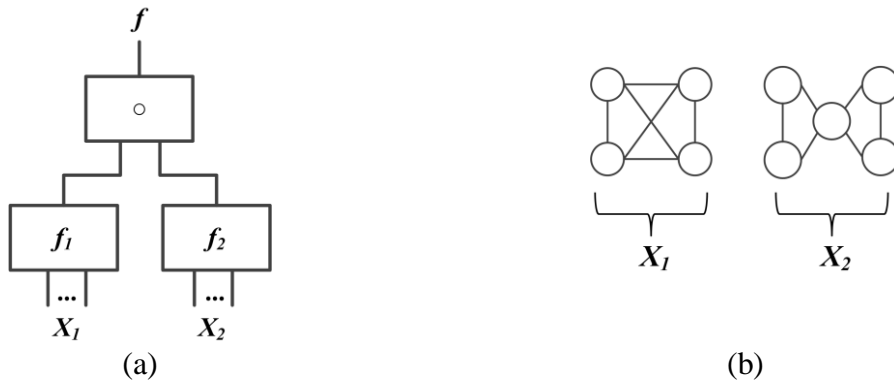


Figure 23 – A generic DSD tree (a) and its corresponding disconnected VIG (b).

By a generalization of Theorem 4 and Theorem 5, if  $f$  is decomposed as in Eq. (13) and is realized through an AND (XOR) operator, *i.e.*  $\circ = \cdot$  ( $\circ = \oplus$ ), the resulting VIG obtained from an IPOS (ESOP) will be disconnected. The analysis of connected components of VIGs reveals information about which kind of decomposition strategy must be applied in order to obtain DSD realizations.

Given a VIG  $G$ , if  $G$  is disconnected into  $m$  connected components, the function  $f$  can be decomposed into  $m$  subfunctions  $h_0(X_0), \dots, h_{m-1}(X_{m-1})$ , where  $X_0 \dots X_{m-1}$  represent mutually disjoint input sets. Notice that there may be cases such that a subfunction  $h_i$  is realized by a literal function, *e.g.*  $h_i(x) = x$ . A decomposition operator  $\circ \in \{\mathbf{AND}, \mathbf{OR}, \mathbf{XOR}\}$  is determined such that the DSD implementation of  $f$  is obtained by  $f(\circ(h_0(X_0), \dots, h_{m-1}(X_{m-1})))$ . If the function  $f$  can be realized by a DSD structure, applying this procedure recursively to subfunctions  $h_0, \dots, h_{m-1}$  will result into a DSD realization of  $f$ . The pseudo-code for the procedure that performs this task is presented in Algorithm 3.

---

**Algorithm 3**


---

**Description** Procedure that receives a set of terms as input, creates and analyze VI graphs.

**Input**  $F_i$ : A set of terms representing  $f$ .  
 $\circ$ : an operator (AND, OR or XOR)

**Output** A disjoint-decomposition tree representing  $f$  or **NULL** if it is not possible.

---

**ANALYSE\_BOOLEAN\_TERMS**( $F_i, \circ$ )

**begin**

```

1: if ( $F_i$  contains only one term  $t_i$ ) return  $t_i$ 
2:  $G :=$  CREATE_VI_GRAPH( $F_i$ )
3: if ( $G$  is not disconnected)
4:   return NULL
5:  $top_{children} := \emptyset$ 
6: for each (connected component  $X_i$ )
7:    $H_i :=$  SELECT_TERMS( $F_i, X_i$ )
8:    $DSD_i :=$  REC_DSC( $H_i$ )
9:   if ( $DSD_i \neq$  NULL)
10:     $top_{children} := top_{children} \cup DSD_i$ 
11:  else
12:    return NULL
13: return CREATE_OPERATOR_NODE( $\circ, top_{children}$ )

```

**end**

---

The main flow of the proposed algorithm is presented in Algorithm 4. Given an input function  $f$ , the possibility of decomposing  $f$  by an OR decomposition (line 2) is analyzed. If an OR decomposition does not exist, then a possible AND realization is examined (line 5). If neither OR nor AND decompositions efforts are successful, an attempt to decompose  $f$  through an XOR decomposition is performed (line 8). If none of the above decompositions returned a successful result, it means  $f$  cannot be realized through a DSD structure.

#### Algorithm 4

---

**Description** Procedure that decomposes  $F$  based on its ISOP, IPOS or ESOP forms.

**Input**  $F$ : A set of terms representing  $f$ .

**Output** A disjoint-decomposition tree representing  $f$  or **NULL** if it is not possible.

---

**REC\_DSD**( $f$ )

**begin**

```
1:  $F_{ISOP} := \text{OBTAIN\_ISOP}(f)$ 
2:  $DSD_{OR} := \text{ANALYSE\_BOOLEAN\_TERMS}(F_{ISOP}, +)$ 
3: if ( $DSD_{OR} \neq \text{NULL}$ ) return  $DSD_{OR}$ 
4:  $F_{IPOS} := \text{OBTAIN\_IPOS}(f)$ 
5:  $DSD_{AND} := \text{ANALYSE\_BOOLEAN\_TERMS}(F_{IPOS}, \cdot)$ 
6: if ( $DSD_{AND} \neq \text{NULL}$ ) return  $DSD_{AND}$ 
7:  $F_{ESOP} := \text{OBTAIN\_ESOP}(f)$ 
8:  $DSD_{XOR} := \text{ANALYSE\_BOOLEAN\_TERMS}(F_{ESOP}, \oplus)$ 
9: if ( $DSD_{XOR} \neq \text{NULL}$ ) return  $DSD_{XOR}$ 
10: return NULL
```

**end**

---

### 5.2.3 A complete example of the proposed approach

The proposed method presented in Algorithm 4 will now be described through a complete example. Let a Boolean function  $f$  be defined as following:

$$\begin{aligned} f(a, b, c, d, e) = & !a!bc!de + !a!bcd!e + !a!bcde + !ab!c!de + !ab!cd!e + !ab!cde + a!b!c!de \\ & + a!b!cd!e + a!b!cde + a!bc!de + a!bcd!e + a!bcde + ab!c!de + ab!cd!e \\ & + ab!cde + abc!de + abcd!e + abcde. \end{aligned}$$

A complete execution tree of the proposed algorithm is depicted in Figure 24, where solid edges denote recursive calls and dotted edges represent the information returned by the successful recursion calls.

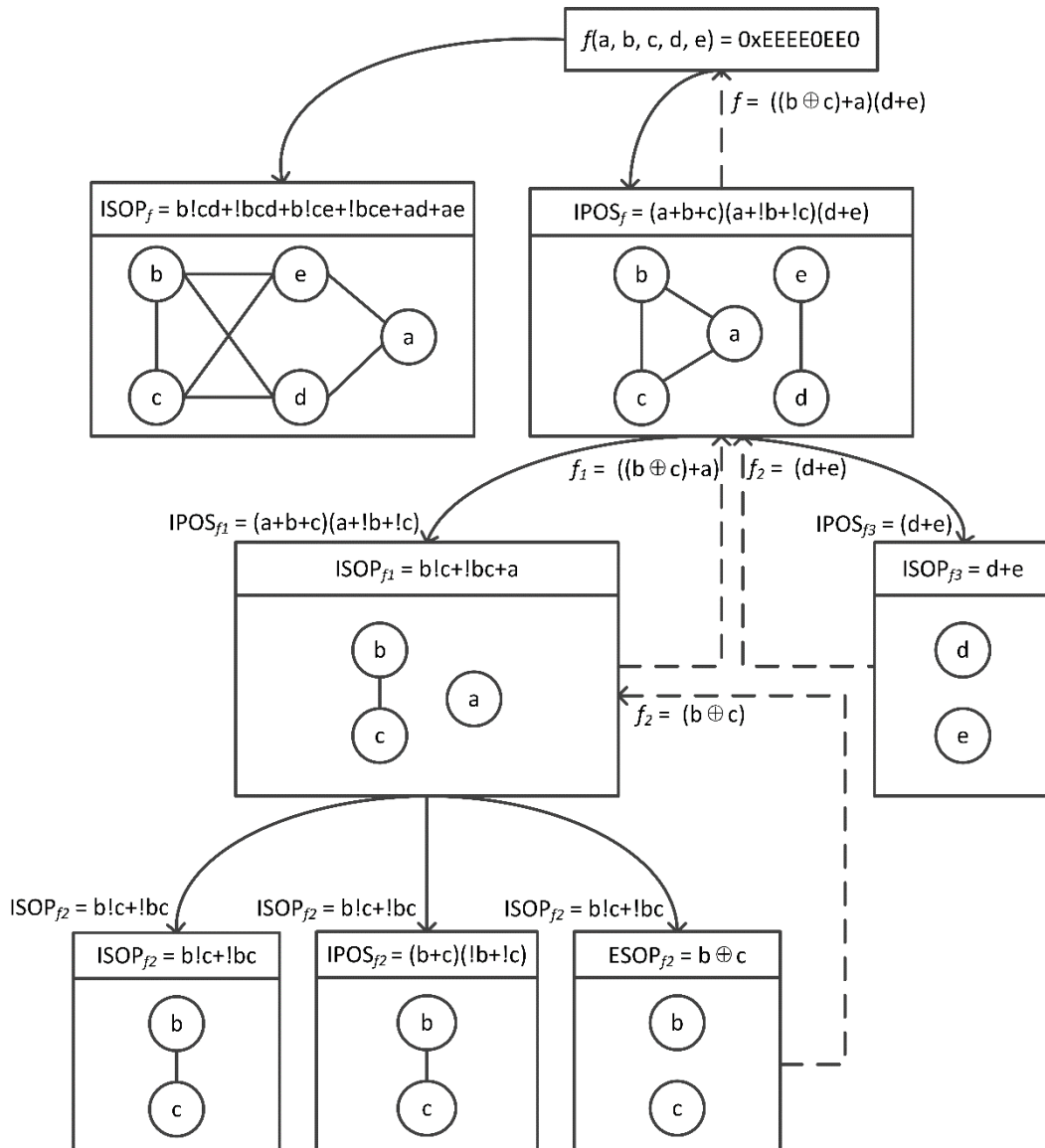


Figure 24 – A complete execution tree of the proposed algorithm.

#### 5.2.4 Experimental results

A framework for synthesis of DSD functions is presented. The proposed framework uses *ESPRESSO* (BRAYTON ET AL, 1984) and *EXORCISM-4* (MISHCHENKO; PERKOWSKI, 2001) tools in order to provide ISOP, IPOS and ESOP forms, respectively. The platform used to perform these results was a Linux system with Intel Core i5 2400 processor and 4GB main memory.

In order to demonstrate the efficiency and accuracy of the proposed approach, two experiments were carried out. The first used all DSD functions with up to 6-inputs. The second experiment was performed over a selection of PLA benchmarks (*ESPRESSO BOOK EXAMPLES*). In all these experiments we are taking into account all the runtime spent

reading and writing files as well as the context switching between calling ESPRESSO and EXORCISM-4 tools.

We grouped all DSD functions with up to 6-inputs into classes by equivalence through input permutation (P-classes). Results comprising the decompositions of all these functions are presented in Table 10. The CPU column shows the time in seconds taken for the decomposition of all functions. Our method successfully decomposed all tested functions.

Table 10 – Results for a benchmark composed of all DSD functions up to 6 inputs, grouped by equivalence through input permutation (P-classes).

Inputs	P classes	CPU (s)	Worst CPU (s)	Avg. CPU (s)
2	8	0.2	0.06	0.02
3	36	1.5	0.11	0.04
4	206	13	0.18	0.06
5	1,259	107	0.27	0.09
6	8,448	909	0.31	0.11

A second experiment was performed over a selection of ESPRESSO book examples. Results are shown in Table 11, where the second and third columns denote the number of primary inputs and the number of primary outputs, respectively. “Decomposed outputs” report the number of outputs that were successfully realized through DSD structures. The column “CPU” reports the time in seconds taken for decompositions.

Table 11 – Results of decompositions over ESPRESSO book PLA benchmark.

Circuit	Primary inputs	Primary outputs	Decomposed outputs	CPU (s)
5xp1	7	10	4	0.97
9sym	9	1	0	0.07
alu4	14	8	0	0.71
apex1	45	45	12	3.37
apex2	39	3	0	3.09
apex3	54	50	18	3.31
apex4	9	19	1	1.21
apex5	117	88	9	7.13
b12	15	9	2	0.52
bw	5	28	7	1.51
clip	9	5	0	0.24
con1	7	2	0	0.08
cordic	23	2	0	5.05
cps	24	109	51	9.02
duke2	22	29	8	2.08
e64	65	65	65	0.88
ex1010	10	10	0	0.77
ex4	128	28	14	1.36
ex5	8	63	35	1.91
inc	7	9	2	0.57
misex1	8	7	0	0.35
misex2	25	18	12	0.7
misex3	14	14	0	1.02
misex3c	14	14	0	0.8
mytest	2	1	1	0.01
pdc	16	40	12	2.48
rd53	5	3	1	0.13
rd73	7	3	1	0.14
rd84	8	4	2	0.17
sao2	10	4	0	0.4
seq	41	35	2	5.03
spla	16	46	12	3.76
squar5	5	8	3	0.41
t481	16	1	1	0.48
table3	14	14	0	1.09
table5	17	15	0	1.62
xor5	5	1	1	0.04
Z5xp1	7	10	4	0.83
Z9sym	9	1	0	0.05
ex1010	10	10	0	1.06
ex4	128	28	14	1.47
ibm	48	17	0	0.82
jbp	36	57	31	3.71
mainpla	27	54	0	5.43
misg	56	23	22	0.35
mish	94	43	43	0.89
misj	35	14	14	0.37
pdc	16	40	12	2.5
shift	19	16	1	0.69
signet	39	8	4	7.6
soar	83	94	47	7.19
test2	11	35	0	5.17
test3	10	35	0	2.52
ti	47	72	21	5.34
ts10	22	16	0	0.72
x7dn	66	15	0	1.27
xparc	41	73	11	13.87



### 5.2.5 Conclusions

This section presented a top-down algorithm for synthesis of simple disjoint-support decompositions (DSD). The algorithm is based on variable intersection graphs which are directly obtained from ISOP, IPOS or from ESOP forms. From our experiments, our approach has successfully synthesized all known DSD functions when tested over the set of all DSD functions with up to 6 inputs as well as over a selection of PLA benchmarks. As future work, the dependency of minimization tools on ESPRESSO and EXORCISM could be removed. Simpler methods can be focused only on the identification and minimization of DSD functions by taking advantage of intrinsic characteristics of DSD functions.

## 5.3 Bottom-up decomposition based on Boolean difference and cofactor analysis

This section presented a new approach to Boolean decomposition based on the Boolean difference and cofactor analysis (CALLEGARO ET AL, 2015b). Two simple tests provide sufficient and necessary conditions to identify AND and exclusive-OR (XOR) decompositions. Such tests were proposed by Kodandapandi, in (KODANDAPANDI; SETH, 1978), in the context of decomposition charts. We revisit such tests, providing a new and efficient cofactor-based approach to obtain decomposition functions more efficiently. Moreover, we extend the decomposition types by providing sufficient and necessary conditions to obtain MUX decompositions with an arbitrary number of inputs. Finally, we present an algorithm that needs  $O(n \cdot \log n)$  cofactor and  $O(n)$  equivalence test operations to perform AND and XOR decomposition. Experimental results have demonstrated the efficiency of the proposed method when compared to the state-of-the-art decomposition strategies.

### 5.3.1 Definitions and notation

Upper case letters  $F$ ,  $G$ ,  $H$  are used to represent functions while variable sets are represented by  $X$ ,  $D$  and  $S$ . A multiplexer (MUX) with  $s$  selectors and  $d$  data inputs ( $d = 2^s$ ) is denoted by:

$$\begin{aligned} MUX(S, D) = & \overline{x_{s_0}} \cdots \overline{x_{s(s-1)}} \cdot x_{D_0} + \dots \\ & + x_{s_0} \cdots x_{s(s-1)} \cdot x_{D(d-1)} \end{aligned} \quad (15)$$

where  $S$  represents the set of MUX selectors,  $D$  represents the set of MUX data inputs, each  $x_{S_i} \in S$  and each  $x_{D_j} \in D$ .

Given a function  $F(X)$ , the *cofactor operation* consists of assigning a value  $c_i \in B$  to an input variable  $x_i \in X$ , which is denoted by  $F^{x_i=c_i}$ . In this section, we also use the notation  $F(x_0, \dots, c_i, \dots, x_{n-1})$  to represent a cofactor operation in  $x_i$ . The *negative (positive) cofactor* with respect to (w.r.t) a variable  $x_i$  is denoted by  $F^{x_i=0}$  ( $F^{x_i=1}$ ). A *cube-cofactor* operation consists of applying cofactors recursively, denoted by  $(F^{x_i=c_i})^{x_j=c_j} = F^{x_i=c_i, x_j=c_j}$ . We also represent a cube-cofactor operation as  $F(x_0, \dots, c_i, \dots, c_j, \dots, x_{n-1})$ . Some cofactors and cube-cofactors identities are presented as follows:

$$F^{x_i=c_i, x_j=c_j} = F^{x_j=c_j, x_i=c_i} \quad (16)$$

$$(F \cdot G)^{x_i=c_i} = F^{x_i=c_i} \cdot G^{x_i=c_i} \quad (17)$$

$$(F \oplus G)^{x_i=c_i} = F^{x_i=c_i} \oplus G^{x_i=c_i} \quad (18)$$

$$x_i^{x_j=c_j} = x_i \quad (19)$$

For presentation sake, the notation described in (20) will be used to represent positive and negative literals w.r.t a variable  $x_i$ . By using these notations, we express both positive and negative literals as well as cofactors and cube-cofactors by assigning values to  $c_i$ . When the value of  $c_i$  is known, we use  $x_i$  and  $\bar{x}_i$  to express positive and negative literals, respectively.

$$x_i^{c_i} = \begin{cases} \bar{x}_i, & \text{if } c_i = 0 \\ x_i, & \text{if } c_i = 1 \end{cases} \quad (20)$$

The *Boolean difference* of  $F$  w.r.t a variable  $x_i$ , is denoted by  $F_{x_i}$  and defined as follows:

$$F_{x_i} = F^{x_i=0} \oplus F^{x_i=1}. \quad (21)$$

Given a function  $F$ , it is said that  $F$  is *dependent* on a variable  $x_i$  iff  $F_{x_i} \neq 0$ . Otherwise,  $F$  is *independent* of  $x_i$ . Constants functions (0 or 1) are independent of any variables. Without loss of generality, we consider that all functions are dependent on all its input variables.

Given a function  $F(X) = H(G(X_1), X_2)$ , where  $X_1 \cap X_2 = \emptyset$ , the Boolean difference w.r.t a variable  $x_i \in X_1$  can be obtained through the *Boolean chain rule* formulation:

$$H_{x_i} = H_G \cdot G_{x_i}, \quad (22)$$

where  $H_G$  is the Boolean difference of  $H$  w.r.t to the subfunction  $G$ .

A Boolean function can be represented by its *Shannon expansion* w.r.t. a variable  $x_i$  as follows:

$$F(X) = \overline{x_i} \cdot F^{x_i=0} + x_i \cdot F^{x_i=1}. \quad (23)$$

The Shannon expansion could be written without defining the polarity applied to both variables and cofactors:

$$F(X) = x_i^{c_i} \cdot F^{x_i=\overline{c_i}} + x_i^{\overline{c_i}} \cdot F^{x_i=c_i}. \quad (24)$$

Notice that equation (24) reduces to equation (23) for any  $c_i$ .

A Boolean function can also be expressed through its Davio expansion (DAVIO; DESCHAMPS; THAYSE, 1978). The *positive Davio expansion* w.r.t a variable  $x_i$  is defined as:

$$F(X) = F^{x_i=0} \oplus x_i \cdot F_{x_i}. \quad (25)$$

Below we present some identities regarding Boolean differential analysis (AKERS, 1959).

$$F_{x_i} = (\overline{F})_{x_i} \quad (26)$$

$$(0)_{x_i} = (1)_{x_i} = 0 \quad (27)$$

$$(x_i)_{x_j} = 0 \quad (28)$$

$$F_{x_i x_j} = F_{x_j x_i} \quad (29)$$

$$F_{x_i x_i} = 0 \quad (30)$$

$$(F^{x_i=c})_{x_j} = (F_{x_j})^{x_i=c} \quad (31)$$

$$(F \oplus G)_{x_i} = F_{x_i} \oplus G_{x_i} \quad (32)$$

$$(F \cdot G)_{x_i} = (F \cdot G_{x_i}) \oplus (F_{x_i} \cdot G) \oplus (F_{x_i} \cdot G_{x_i}) \quad (33)$$

The representation of a function into subfunctions is called functional decomposition. A DSD function can be represented by  $F(X) = H(G(X_1), X_2)$  such that  $X_1 \cap X_2 = \emptyset$ , where  $G$  and  $H$  are called composition and decomposition functions of  $F$ , respectively. The variable set  $X_1$  and  $X_2$  are named bound and free sets, respectively. A pictorial representation of DSD is shown in Figure 25. Notice that, by definition, constants and input variables (and its complement) are DSD.

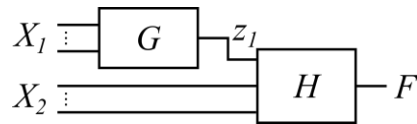


Figure 25 – Illustration of  $F(X) = H(G(X_1), X_2)$ . Functions  $G$  and  $H$  are called composition and decomposition functions of  $F$ , respectively. The variable set  $X_1$  and  $X_2$  are named bound and free sets, respectively.

A *full-DSD* function is a constant, a variable (or its complement), or a composition of full-DSD functions with disjoint support. A function is *partial-DSD* if either its decomposition or its composition function is full-DSD, but not both. A *non-DSD* function is neither a full-DSD nor a partial-DSD function. For instance, the majority (MAJ) function, also known as voter, is non-DSD.

The decomposition can be performed on the outputs to the inputs (*top-down*) or from the inputs to the outputs (*bottom-up*). Both strategies can synthesize full-DSD functions. For partial-DSD functions, both strategies can be combined. Top-down approaches can identify DSD decomposition functions, as shown in Figure 26 (a), while bottom-up approaches are suitable for identifying composition functions, as depicted in Figure 26 (b).

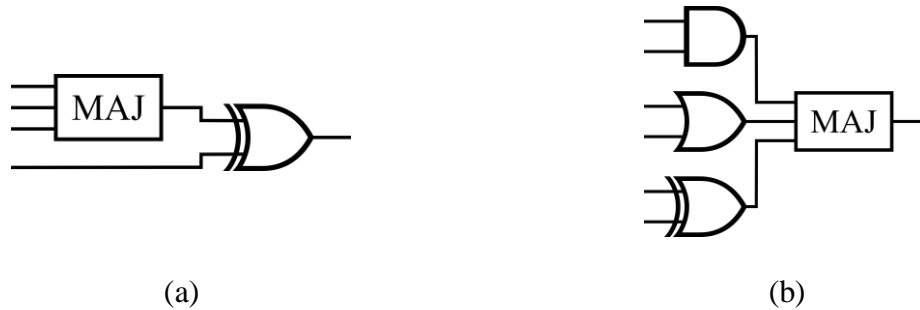


Figure 26 – Two examples of partial-DSD functions. A top-down approach can find the XOR decomposition on top of (a), while a bottom-up approach could not. In (b), a bottom-up approach identifies the AND, OR and XOR compositions, while a top-down approach finds nothing.

### 5.3.2 Bottom-Up Decomposition Properties

Most DSD methods consider only AND and XOR as basic gates. This interest is because all Boolean functions with up to two inputs are DSD considering such a base. For instance, the OR operation can be performed by using only AND and complement. Moreover, AND (XOR) functions with an arbitrary number of inputs can be obtained by DSD representations of two inputs ANDs (XORs).

Some recently proposed decomposition approaches also consider 2-to-1-MUXes as basic gate (MISHCHENKO; BRAYTON, 2013), (MISHCHENKO, 2014). However, DSD representations of MUXes with an arbitrary number of inputs cannot be achieved with AND, XOR, and 2-to-1 MUX as basis. For example, the circuit depicted in Figure 27 (a) is not full-DSD when considering AND, XOR, and 2-to-1 MUX as basis. The circuit depicted in Figure 27 (b) is partial-DSD for top-down decompositions since the AND decomposition is identified in the output. However, the 16-to-1 MUX is not found. Both circuits are full-DSD when considering as basis, AND, XOR and MUX with an arbitrary number of inputs.

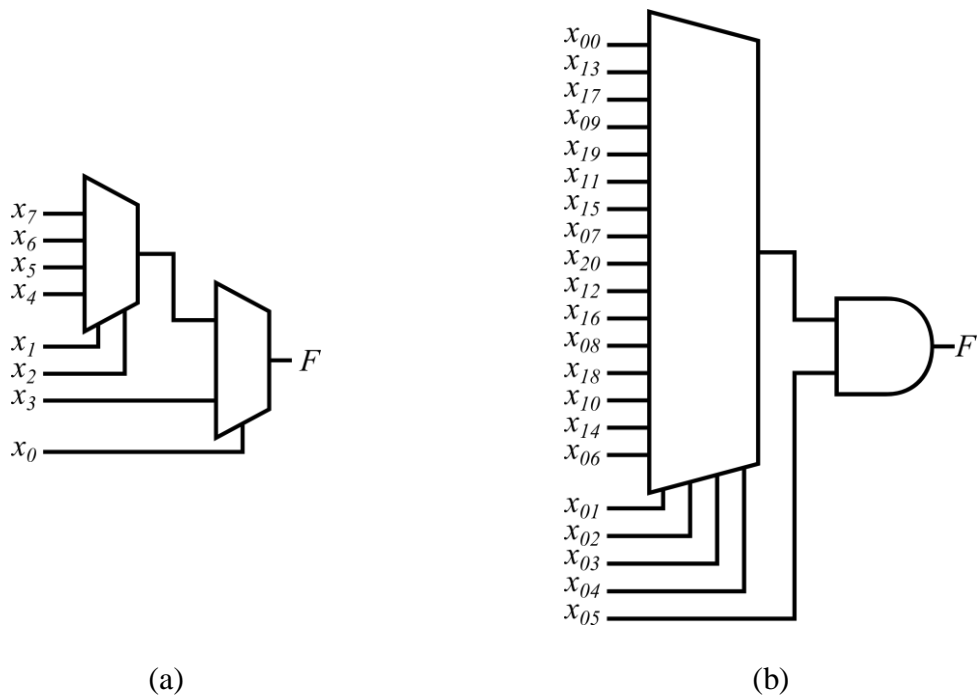


Figure 27 – Two examples of full-DSD functions when considering AND, XOR and MUX as basis. The example shown in (a) is the implementation of the output 04 of the *shift* circuit, taken from the (ESPRESSO BOOK EXAMPLES). The circuit depicted in (b) is the implementation of the *mux* circuit, present in the ACM/SIGDA (MCNC) benchmark (IWLS, 2005).

In (KODANDAPANDI; SETH, 1978), necessary and sufficient conditions to perform AND and XOR bottom-up decompositions were presented. The authors proposed a strategy to obtain decomposition functions through the analysis of decomposition charts. We revisit such conditions, providing a faster method based on cofactors to obtain such decomposition functions. This strategy is presented in Corollary 1 and 2. Furthermore, we extend the decomposition types by providing sufficient and necessary conditions to obtain MUX decompositions with an arbitrary number of inputs. These conditions are presented in

Theorems 8 and 4. The decomposition functions for MUX decomposition are presented in Corollaries 3 and 4.

### 5.3.3 AND decomposition

**Theorem 6.** Let  $F(X_I, X_2)$  be a Boolean function with  $X_I = \{x_i, x_j\}$ ,  $X_I \cap X_2 = \emptyset$ . There exists a function  $H(z_I, X_2)$ , where  $z_I = G(X_I) = x_i^{c_i} \cdot x_j^{c_j}$ , such that  $F(X_I, X_2) = H(G(X_I), X_2)$ , if and only if:

$$F^{x_i=c_i} = F^{x_j=c_j} \quad (34)$$

**Proof.** See (KODANDAPANDI; SETH, 1978) ■

**Corollary 1.** If there exists a function  $H(z_I, X_2)$ , where  $z_I = G(X_I) = x_i^{c_i} \cdot x_j^{c_j}$  such that  $F(X_I, X_2) = H(G, X_2)$ , then the decomposition function  $H(z_I, X_2)$  can be obtained as follows:

$$H(z_I, X_2) = \bar{z}_I \cdot F^{x_i=c_i} + z_I \cdot F^{x_i=c_i, x_j=c_j} \quad (35)$$

**Proof.** By definition,  $F(x_i, x_j, X_2) = H(G, X_2)$ . Then  $(x_i = \bar{c}_i) \rightarrow (G^{x_i=c_i} = 0) \rightarrow H(0, X_2) = F(\bar{c}_i, x_j, X_2)$  and  $(x_i = c_i, x_j = c_j) \rightarrow (G^{x_i=c_i, x_j=c_j} = 1) \rightarrow H(1, X_2) = F(c_i, c_j, X_2)$ . By applying the Shannon expansion (23) in  $H(z_I, X_2)$ , we obtain:

$$H(z_I, X_2) = \bar{z}_I \cdot H^{z_I=0} + z_I \cdot H^{z_I=1} \quad (36)$$

By substitution of  $H^{z_I=0} = H(0, X_2) = F(\bar{c}_i, x_j, X_2)$  and  $H^{z_I=1} = H(1, X_2) = F(c_i, c_j, X_2)$  in (36), (35) is found. Notice that since  $F^{x_i=c_i} = F^{x_j=c_j}$  (35) could be written using  $x_j$  and  $c_j$  in place of  $x_i$  and  $c_i$ . ■

### 5.3.4 XOR decomposition

**Theorem 7.** Let  $F(X_I, X_2)$  be a Boolean function with  $X_I = \{x_i, x_j\}$ ,  $X_I \cap X_2 = \emptyset$ . There exists a function  $H(z_I, X_2)$ , where  $z_I = G(X_I) = x_i \oplus x_j$ , such that  $F(X_I, X_2) = H(G(X_I), X_2)$ , if and only if:

$$F_{x_i} = F_{x_j} \quad (37)$$

**Proof.** See (KODANDAPANDI; SETH, 1978). ■

**Corollary 2.** If there exists a function  $H(z_l, X_2)$ , where  $z_l = G(X_l) = x_i \oplus x_j$  such that  $F(X_l, X_2) = H(G, X_2)$ , then the decomposition function  $H(z_l, X_2)$  can be obtained as follows:

$$H(z_l, X_2) = F^{x_i=0, x_j=0} \oplus z_l \cdot F_{x_i} \quad (38)$$

**Proof.** By applying the positive Davio expansion (25) of  $H$  in  $z_l$ , we obtain:

$$H(z_l, X_2) = H^{z_l=0} \oplus z_l \cdot H_{z_l} \quad (39)$$

The value of  $H^{z_l=0}$  is obtained by finding an assignment for  $x_i$  and  $x_j$  such that  $G(x_i, x_j) = 0$ , e.g.  $G^{x_i=0, x_j=0} = G^{x_i=1, x_j=1} = 0$ . Since by definition  $F(x_i, x_j, X_2) = H(G, X_2)$ , we then consider  $H^{z_l=0} = F^{x_i=0, x_j=0}$ . The value for  $H_{z_l}$  is obtained as follows.

By applying the Boolean difference (21) of  $G$  in  $x_i$ , we obtain:

$$G_{x_i} = (x_i \oplus x_j)^{x_i=0} \oplus (x_i \oplus x_j)^{x_i=1} \quad (40)$$

and by simplification,

$$G_{x_i} = x_j \oplus \overline{x_j} = 1 \quad (41)$$

is obtained. The same can be found for  $x_j$ , i.e.  $G_{x_j} = 1$ . Through the Boolean chain rule formulation (22), the Boolean difference of  $H$  in  $x_i$  is  $H_{x_i} = H_G \cdot G_{x_i}$ . As shown in (41),  $G_{x_i} = 1$  and through simplification,  $H_{z_l} = H_G = H_{x_i}$ . Since  $F(x_i, x_j, X_2) = H(G, X_2)$ , then  $H_{z_l} = H_{x_i} = F_{x_i}$  (the same can be obtained for  $x_j$ ). By substitution of  $H^{z_l=0} = F^{x_i=0, x_j=0}$  and  $H_{z_l} = F_{x_i}$  into (39), (38) is then found. ■

### 5.3.5 MUX decomposition

Below sufficient and necessary conditions to obtain 2-to-1 MUX decomposition is presented. Without loss of generality, we consider that all MUX input variables are positive.

**Theorem 8.** Let  $F(X_l, X_2)$  be a Boolean function with  $X_l = S \cup D$ ,  $S = \{x_s\}$ ,  $D = \{x_i, x_j\}$ ,  $S \cap D = \emptyset$ ,  $X_l \cap X_2 = \emptyset$ . There exists a function  $H(z_l, X_2)$ , where  $z_l = G(X_l) = MUX(S, D) = \overline{x_s} \cdot x_i + x_s \cdot x_j$ , such that  $F(X_l, X_2) = H(G(X_l), X_2)$ , if and only if:

$$F_{x_i} \neq F_{x_j} \quad (42)$$

$$F_{x_i x_j} = 0 \quad (43)$$

$$(F_{x_i})^{x_s=1} = 0 \quad (44)$$

$$(F_{x_j})^{x_s=0} = 0 \quad (45)$$

$$(F^{x_i=0, x_j=0})_{x_s} = 0 \quad (46)$$

$$(F^{x_i=1, x_j=1})_{x_s} = 0 \quad (47)$$

**Proof.** (if part). By applying the positive Davio expansion (25) of  $F$  in  $x_i$  and  $x_j$  we obtain:

$$\begin{aligned} F(X) &= (F^{x_i=0, x_j=0} \oplus x_i^{x_j=0} \cdot F_{x_i}^{x_j=0}) \\ &\oplus x_j \cdot (F^{x_i=0}_{x_j} \oplus (x_i \cdot F_{x_i})_{x_j}) \end{aligned} \quad (48)$$

By simplifying (48) using (19), (43), (33) and (28), we obtain:

$$F(X) = F^{x_i=0, x_j=0} \oplus x_i \cdot F_{x_i} \oplus x_j \cdot F_{x_j} \quad (49)$$

We then apply the Shannon expansion (24) of (49) in  $x_s$ , and by using (19), (45), (44), (48) and simplifications, we obtain:

$$F(X) = F^{x_i=0, x_j=0} \oplus \overline{x_s} \cdot x_i \cdot F_{x_i}^{x_s=0} \oplus x_s \cdot x_j \cdot F_{x_j}^{x_s=1} \quad (50)$$

By using the equivalence of (46) and (47):

$$(F^{x_i=0, x_j=0})_{x_s} = (F^{x_i=1, x_j=1})_{x_s}, \quad (51)$$

and simplifications using (44) and (45), we show that:

$$F_{x_i}^{x_s=0} = F_{x_j}^{x_s=1} \quad (52)$$

Then, by substitution of (52) in (50), and using  $F_{x_i}^{x_s=0}$  as reference, we obtain:

$$F(X) = F^{x_i=0, x_j=0} \oplus F_{x_i}^{x_s=0} \cdot (\overline{x_s} \cdot x_i \oplus x_s \cdot x_j) \quad (53)$$

Since both  $F^{x_i=0, x_j=0}$  and  $F_{x_i}^{x_s=0}$  are independent of  $x_i, x_j, x_s$ , it follows  $G(X_1) = MUX(S, D) = \overline{x_s} \cdot x_i + x_s \cdot x_j$  is a subfunction of  $F$  and  $F(X_1, X_2) = H(G(X_1), X_2)$ .

(only if part): Consider  $F(X_1, X_2) = H(G(X_1), X_2)$ , where  $G(X_1) = MUX(S, D) = \overline{x_s} \cdot x_i + x_s \cdot x_j$ . The Boolean difference of  $F$  concerning  $x_i, x_j$  can be obtained by the Boolean chain rule formulation (22):



$$F_{x_i} = F_G \cdot G_{x_i} \text{ and } F_{x_j} = F_G \cdot G_{x_j}, \quad (54)$$

where  $G_{x_i} = \overline{x_s}$  and  $G_{x_j} = x_s$ . Clearly,  $F_{x_i} \neq F_{x_j}$  (42). The Boolean difference of  $F_{x_i}$  w.r.t.  $x_j$  is  $F_{x_i x_j} = (F_G \cdot G_{x_i x_j}) \oplus (F_{G_{x_j}} \cdot G_{x_i}) \oplus (F_{G_{x_j}} \cdot G_{x_i x_j})$ , where  $G_{x_i x_j} = 0$  and  $F_{G_{x_j}} = 0$  since both functions do not depend on  $x_j$ . Then,  $F_{x_i x_j} = 0$  (43). In order to obtain the positive cofactor of  $F_{x_i}$  w.r.t  $x_s$ , (17) is applied in (54), resulting  $(F_{x_i})^{x_s=1} = (F_G)^{x_s=1} \cdot (G_{x_i})^{x_s=1}$ . Since  $(G_{x_i})^{x_s=1} = 0$   $(F_{x_i})^{x_s=1} = 0$  (44). The same can be obtained for  $(F_{x_j})^{x_s=0} = 0$  (45). Since  $G_{x_s} = (x_i \oplus x_j)$ ,  $F_{x_s} = F_G \cdot (x_i \oplus x_j)$ . Then, it follows that  $(F_{x_s})^{x_i=0, x_j=0} = (F_{x_s})^{x_i=1, x_j=1} = 0$ . By (31), it is clear that  $(F^{x_i=0, x_j=0})_{x_s} = 0$  (46) and  $(F^{x_i=1, x_j=1})_{x_s} = 0$  (47). ■

**Corollary 3.** If there exists a function  $H(z_l, X_2)$ , where  $z_l = G(X_l) = MUX(S, D) = \overline{x_s} \cdot x_i + x_s \cdot x_j$  such that  $F(X_l, X_2) = H(G, X_2)$ ,  $X_l = S \cup D$ ,  $S = \{x_s\}$ ,  $D = \{x_i, x_j\}$ ,  $S \cap D = \emptyset$ ,  $X_l \cap X_2 = \emptyset$  then the decomposition function  $H(z_l, X_2)$  can be obtained as follows:

$$H(z_l, X_2) = F^{x_i=0, x_j=0} \oplus z_l \cdot F_{x_i}^{x_s=0} \quad (55)$$

*Proof.* By applying the positive Davio expansion (25) of  $H$  in  $z_l$ , we obtain:

$$H(z_l, X_2) = H^{z_l=0} \oplus z_l \cdot H_{z_l} \quad (56)$$

The value for  $H^{z_l=0}$  can be obtained by finding an assignment for  $x_i$  and  $x_j$  such that  $G(x_i, x_j) = 0$ , e.g.  $G^{x_i=0, x_j=0} = 0$ . Then it follows that  $H^{z_l=0} = H(0, X_2) = F^{x_i=0, x_j=0}$ . Since by definition  $F(X_l, X_2) = H(G(x_i, x_j), X_2)$ , we can obtain the Boolean difference of  $F$  w.r.t  $x_i$  using the Boolean chain rule  $F_{x_i} = F_G \cdot G_{x_i}$ , where  $G_{x_i} = \overline{x_s}$ . Then, it follows from (17) that  $F_{x_i}^{x_s=0} = F_G^{x_s=0} \cdot (\overline{x_s})^{x_s=0}$  and therefore  $F_{x_i}^{x_s=0} = F_G$ , since  $F_G$  does not depend on  $x_s$ . Since  $F(X_l, X_2) = H(z_l, X_2)$ , where  $z_l = G$ ,  $F_G = H_{z_l}$  and therefore (55) holds. ■

We now extend Theorem 8, presenting a small set of tests that provide necessary and sufficient conditions to obtain MUX decompositions with an arbitrary number of inputs.

**Theorem 9.** Let  $F(X)$  be a Boolean function and  $S, D$  and  $X_2$  be proper disjoint subsets of  $X$  such that  $S \cup D \cup X_2 = X$ , where  $|D| = 2^{|S|}$ ,  $S = \{x_{s0}, \dots, x_{s|S|-1}\}$ ,  $D = \{x_{d0}, \dots, x_{d|D|-1}\}$ . There

exists a function  $H(z_I, X_2)$ , where  $z_I = G(X_I) = \text{MUX}(S, D)$ , such that  $F(X_I, X_2) = H(G(X_I), X_2)$ , if and only if:

$$(\forall_{x_{d_i}, x_{d_j} \in D})((x_{d_i} \neq x_{d_j}) \rightarrow (F_{x_{d_i}} \neq F_{x_{d_j}})) \quad (57)$$

$$(\forall_{x_{d_i}, x_{d_j} \in D})(F_{x_{d_i}x_{d_j}} = 0) \quad (58)$$

$$(\forall_{x_{d_i} \in D})(\forall_{x_{s_j} \in S})(\exists! c \in B) | (F_{x_{d_i}}^{x_{s_j} = c} = 0) \quad (59)$$

$$\begin{aligned} & (\forall x_s \in S)(\exists c_{d_0}, c_{d_1}, \dots, c_{d_{|D|-1}} \in B^{|D|}) | \\ & (((F^{x_{d_0}=c_{d_0}, x_{d_1}=c_{d_1}, \dots, x_{d_{|D|-1}}=c_{d_{|D|-1}}})_{x_s} = 0) \wedge \\ & ((F^{x_{d_0}=\overline{c_{d_0}}, x_{d_1}=\overline{c_{d_1}}, \dots, x_{d_{|D|-1}}=\overline{c_{d_{|D|-1}}}})_{x_s} = 0)) \end{aligned} \quad (60)$$

*Proof sketch (if and only if):* This theorem is a generalization of Theorem 8 for multiplexers with an arbitrary number of inputs, and therefore, having a similar proof. Notice that (57) to (60) generalize (42) to (47). In order to obtain such proof, apply the positive Davio expansion (25) for each element in  $D$  then apply the Shannon expansion (24) for all elements in  $S$ . Finally, use (57) to (60) to simplify the equation. ■

**Corollary 4** . If there exists a function  $H(z_I, X_2)$ , where  $z_I = G(X_I) = \text{MUX}(S, D)$ ,  $F(X_I, X_2) = H(G, X_2)$  such that  $X_I = S \cup D$ ,  $X_I \cup X_2 = X$ ,  $X_I \cap X_2 = \emptyset$ ,  $|D| = 2^{|S|}$ ,  $S = \{x_{s_0}, \dots, x_{s_{|S|-1}}\}$ ,  $D = \{x_{d_0}, \dots, x_{d_{|D|-1}}\}$ ,  $S \cap D = \emptyset$ , then the decomposition function  $H(z_I, X_2)$  can be obtained as follows:

$$H(z_I, X_2) = F_{z_I}^{x_{d_0}=\overline{c_{d_0}}, x_{d_1}=\overline{c_{d_1}}, \dots, x_{d_{|D|-1}}=\overline{c_{d_{|D|-1}}}} \oplus z_I \cdot F_{x_I}^{x_{s_0}=\overline{c_{s_0}}, \dots, x_{s_{|S|-1}}=\overline{c_{s_{|S|-1}}}} \quad (61)$$

*Proof sketch (if and only if):* This corollary is a generalization of Corollary 3 for multiplexers with an arbitrary number of inputs, and therefore, having a similar proof. ■

### 5.3.6 Proposed full-DSD synthesis method

We now present algorithms to identify and synthesize DSD functions. The methods decompose a function  $F$  strictly from the inputs to the outputs, i.e. bottom-up decomposition.

#### 5.3.6.1 Trivial implementation

The first method is a trivial implementation of Theorems 1 and 2 and Corollaries 1 and 2. When a decomposition is found, a new function  $F'$  is composed, reducing the variable support

in one unit. This process is performed until no more decompositions are found. This method requires  $O(n^3)$  equivalence tests and cofactor operations.

---

### Algorithm 5

---

**Description** Trivial implementation of the proposed DSD method.

**Input**  $F$ : A Boolean function.

**Output** A disjoint-decomposition tree representing  $F$  or **NULL** if it is not possible.

---

**TRIVIAL\_DSD** ( $F$ )

**begin**

1:  $F' = F$

2:  $X' = X$

3: dec = **true**

4: **while** (dec)

5: dec = false

6: **for** ( $i = 0$ ; !dec and  $i < |X'|$ ;  $i++$ )

7: **for** ( $j = i+1$ ; !dec and  $j < |X'|$ ;  $j++$ )

8: **if** (*test\_for\_and\_dec*( $x_i, x_j$ )) // Theorem 6

9: dec = **true**

10:  $z_k = x_i^{c_i} \cdot x_j^{c_j}$

11:  $F' = \text{compose } H \text{ according Corollary 1}$

12:  $X' = (X \setminus \{x_i, x_j\}) \cup z_k$

13: **else if** (*test\_for\_xor\_dec*( $x_i, x_j$ )) // Theorem 7

14: dec = true

15:  $z_k = x_i \oplus x_j$

16:  $F' = \text{compose } H \text{ according Corollary 2}$

17:  $X' = (X \setminus \{x_i, x_j\}) \cup z_k$

18: **if** ( $|X'| > 1$ ) **return**  $F'$  as partial- or non-DSD properly

19: **else if** ( $F^{x_i=1} == \text{constant zero}$ ) **return**  $\overline{x_0}$

20: **else return**  $x_0$

**end**

---

### 5.3.6.2 Lazy evaluation of decomposition functions

Consider  $H(G(X_1), X_2)$  a decomposition of  $F(X_1, X_2)$ . Instead of evaluating function  $H$ , this method obtains negative ( $F^{G=0}$ ) and positive cofactors ( $F^{G=1}$ ) as well as Boolean difference ( $F^{G=0} \oplus F^{G=1}$ ) for a subfunction  $G$  without reducing the function support. On the one hand, evaluating decomposition functions leads to support reduction, which implies faster equivalence test and cofactor operations. On the other hand, by evaluating positive and negative cofactors w.r.t the found subfunctions, cofactors and the Boolean difference already evaluated can be reused. We choose not to reduce the support, thus avoiding unnecessary cofactors operations.

Negative and positive cube-cofactors of a subfunction  $G$  of  $F$  are obtained considering the smallest possible number of cofactors needed to make  $G$  assume a constant value. For example, suppose there exists an AND decomposition of  $F$  represented by  $G(G_1, G_2) = G_1 \cdot G_2$ , where  $G_1$  and  $G_2$  are subfunctions of  $G$ . Consider also that negative and positive cofactors for both  $G_1$  and  $G_2$  have already been calculated and stored.  $F^{G=0}$  is directly obtained from  $F^{G_1=0}$  or  $F^{G_2=0}$ . However, in order to obtain  $F^{G=1}$ , a cube-cofactor is necessary. The positive cofactor of  $G$  can be obtained by  $(F^{G_1=1})^{G_2=1}$  or  $(F^{G_2=1})^{G_1=1}$ . Considering that the number of cube-cofactor operations for performing the positive cofactor of  $G_1$  is larger than  $G_2$ , we choose to reuse the  $F^{G_1=1}$  as basis when applying the  $G_2$  positive cofactor, resulting  $(F^{G_1=1})^{G_2=1}$ . This process is also applied to XOR decompositions.

The pseudocode of our algorithm is presented in Algorithm 6. The worst case runtime of the algorithm is when the given function is full-DSD, as presented in Figure 28. The algorithm starts by evaluating negative and positive cofactors for all input variables (line 2 – Algorithm 6). For each node  $n_i$  in level  $d$ , there are  $2^{d-1}$  input variables (leaf nodes) that  $n_i$  depends on. In order to obtain negative and positive cube-cofactors for  $n_i$ , at most  $2^{d-2}$  cofactors are performed. This can be achieved by reusing an already calculated cofactor of a subfunction of  $n_i$ . In the worst case, such an operation is applied for each non-leaf node of the tree. Since there are  $n / 2^{d-1}$  nodes at level  $d$ , the number of cofactors necessary to synthesize a full-DSD function using Algorithm 6 is obtained as follows:

$$2n + \sum_{i=2}^{\lceil \log_n \rceil + 1} 2 \cdot 2^{i-2} \cdot \frac{n}{2^{i-1}}. \quad (62)$$

---

**Algorithm 6**

---

**Description** Lazy implementation of the proposed DSD method.

**Input**  $F$ : A Boolean function.

**Output** A disjoint-decomposition tree representing  $F$  or **NULL** if it is not possible.

---

**LAZY\_DSD** ( $F$ )

**begin**

```
1:  $N = \text{obtain\_dsd\_nodes\_from\_inputs}(X)$ 
2:  $n = |N|$ 
3: if ( $n == 0$ ) return ( $F == \text{constant zero} ? \text{"0"} : \text{"1"}$ )
4:  $M = \text{create a DSD node array with } n \text{ empty positions}$ 
5:  $m = 0$ 
6: while ( $n > 0$ )
7:    $n' = 0$ 
8:   for ( $i = 0; i < n; i++$ )
9:      $n_i = N[i], n_k = \text{null}$ 
10:    for ( $j = 0; j < m;$ )
11:       $n_j = M[j]$ 
11:      if ( $\text{test\_for\_and\_dec}(n_i, n_j)$ ) // Theorem 6
13:         $n_k = \text{create\_dsd\_node}(\text{AND}, n_i, n_j)$ 
14:      else if ( $\text{test\_for\_xor\_dec}(n_i, n_j)$ ) // Theorem 7
15:         $n_k = \text{create\_dsd\_node}(\text{XOR}, n_i, n_j)$ 
16:      if ( $n_k \neq \text{null}$ )
17:         $M[j] = M[--m]$ 
18:      break;
19:    else  $j++$ 
20:    if ( $n_k \neq \text{null}$ )  $N[n'++] = n_k$ 
21:    else  $M[m++] = n_i$ 
22: if ( $m == 1$ )
23:    $n_s = M[0]$  // node  $n_s$  contains a full-DSD solution for  $F$ 
24:   if ( $F^{n_s=0} \neq \text{constant zero}$ ) return  $\overline{n_s}$ 
25:   else return  $n_s$ 
26:  $H = \text{obtain decomposition function}$ 
27: return  $H$ 
```

**end**

---

Therefore, the worst case on the number of cofactors is  $2n + n \log n$  and consequently  $O(n \log n)$ . The number of equivalence test operations necessary to synthesize an  $n$ -input full-DSD function is  $O(n^2)$ . However, Algorithm 6 can be modified to store cofactors and Boolean difference functions in a hash table, reducing the number of equivalence test operations. Considering a perfect hash function, only one equivalence test is performed for each decomposition found. In this sense, the worst case number of equivalence test operations is  $O(n)$ .

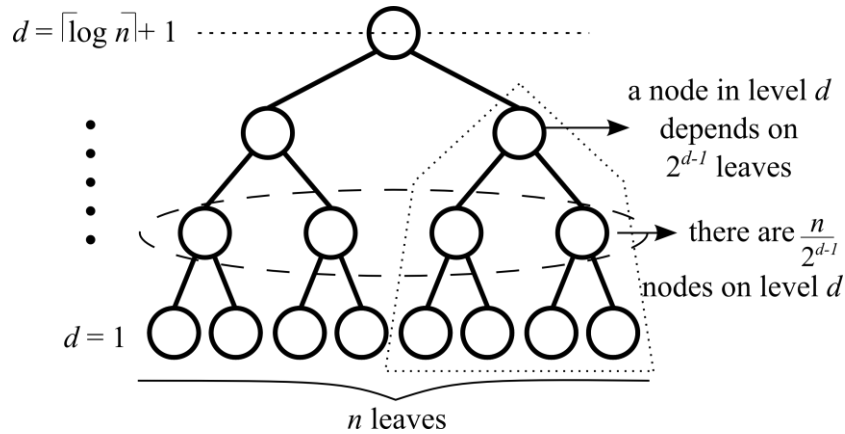


Figure 28 – A full-tree model, representing the worst-case of a full-DSD function.

### 5.3.7 Experimental Results

This section presents results of the proposed algorithm. Algorithm 6 was implemented in the ABC framework (BERKELEY, 2016). We use the ABC truth table data structure to represent Boolean functions. Hence, the method can efficiently manipulate functions with up to 16 inputs. The platform used to perform these results was a Linux system with Intel Core i5 2400 processor and 4GB main memory.

We compared our approach to the state-of-the-art algorithms available in the ABC tool. The methods can be executed by the commands `testdec -A 3` (MISHCHENKO; STEINBACH; PERKOWSKI, 2001) and `testdec -A 4`, an improved implementation of the method presented in (MISHCHENKO; STEINBACH; PERKOWSKI, 2001). We compared the methods by applying them over various benchmarks of DSD functions. All methods obtained DSD solutions for all examples.

The first experiment was carried out over an enumeration of full-DSD functions (CALLEGARO, 2016). The first set of functions comprises all full-DSD functions with up to

6 inputs, as shown in the first row in Table 12. The second row represents a random subset of full-DSD functions with 7 inputs. The proposed method is more efficient than the algorithm presented in (MISHCHENKO; STEINBACH; PERKOWSKI, 2001), considering both the original and improved versions.

Table 12 – Comparison of DSD methods considering two sets of full-DSD functions.

Inputs	Functions	<i>testdec -A 3</i>	<i>testdec -A 4</i>	Our method
6	2,311,640	5.49 sec	6.76 sec	1.5 sec
7	2,744,691	10.7 sec	10.15 sec	2.64 sec

The second experiment was carried out over the same benchmarks taken into account in (HUANG ET AL, 2013). The results presented in Figure 29 compare the CPU time (in seconds) when running each method over each benchmark.

The proposed method is faster for functions with up to 10 inputs. For functions with more than 12 inputs, the method described in (MISHCHENKO; STEINBACH; PERKOWSKI, 2001) with improvements presents a shorter runtime. This behavior is due to the fact that the method in (MISHCHENKO; STEINBACH; PERKOWSKI, 2001) also performs top-down decomposition. This strategy can speed up the algorithm. Currently, the proposed method uses only bottom-up decompositions. It is expected that the addition of a top-down decomposition in our method will reduce its execution time even more. As future work, we intend to merge top-down and bottom-up decompositions using a hash table to store information.

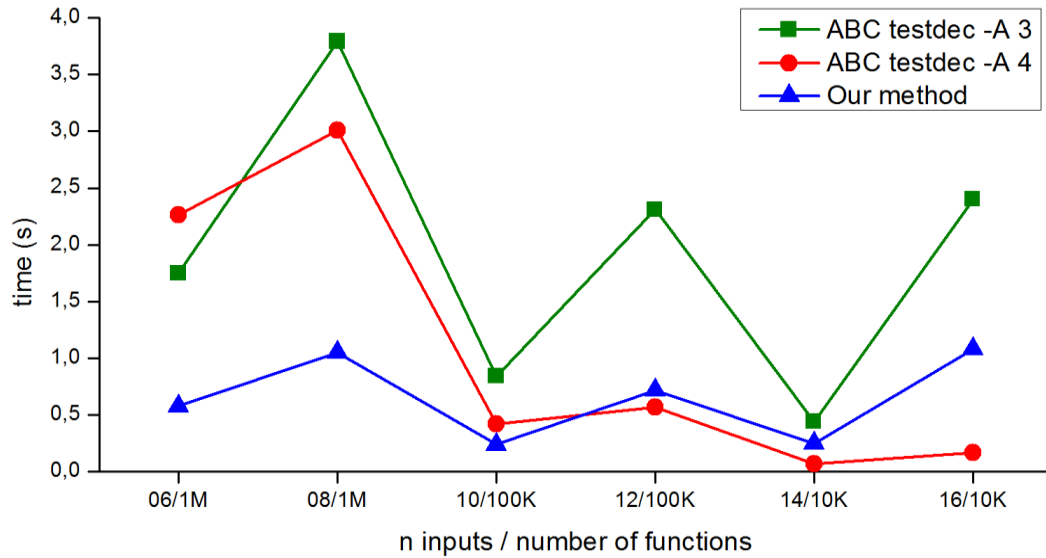


Figure 29 – Comparison of the state-of-the-art algorithms available in ABC tool on a full-DSD benchmark.

### 5.3.8 Conclusions

In this section, a new approach to Boolean decomposition based on the Boolean difference and cofactor analysis was presented. Moreover, we extend the decomposition types by providing sufficient and necessary conditions to obtain MUX decompositions with an arbitrary number of inputs. Finally, we present an algorithm that needs  $O(n \cdot \log n)$  cofactor and  $O(n)$  equivalence test operations to perform AND and XOR decomposition. Experimental results have demonstrated the efficiency of the proposed method when compared to the state-of-the-art decomposition strategies.



## 6 READ-POLARITY-ONCE FUNCTIONS

In VLSI design, the logic synthesis process includes the minimization of a circuit cost function which is independent of the final technology adopted. Typically the cost function in such step is the total literal count of the factored representation of the target logic function (which correlates quite well with circuit area) (BRAYTON, 1987), (HACHTEL; SOMENZI, 2006).

There are several algorithms proposed for factoring Boolean functions, including algebraic (BRAYTON, 1987), (SENTOVICH ET AL, 1992), Boolean (YOSHIDA; FUJITA, 2011) (MARTINS ET AL, 2012) and graph-based approaches (MINTZ, GOLUMBIC, 2005). However, none of these algorithms guarantees the minimum literal count for general Boolean functions, since this is an NP-hard problem (HACHTEL; SOMENZI, 2006). Optimality can be easily checked and guaranteed only for special classes of functions, like as read-once (HAYES, 1975), disjoint support decomposition (ASHENHURST, 1957), (CURTIS, 1962) and read-polarity-once Boolean functions (CALLEGARO ET AL, 2012).

Efficient exact algorithms exist for a sub-class of functions known as read-once functions (GOLUMBIC; MINTZ; ROTICS, 2001), (GOLUMBIC; MINTZ; ROTICS, 2008). There is another class of Boolean functions that can be synthesized and proved minimal, known as disjoint support decomposition (DSD) Boolean functions. Similar to RO functions, a DSD function can be expressed in a factored form where each variable appears only once. However, besides the *NOT* (!), *AND* (\*) and *OR* (+) operators, DSD functions can also use the *XOR* ( $\oplus$ ) operator, e.g.  $g = (a+b)\oplus(c*d+e)$ . Clearly RO functions are a subset of DSD functions. Unfortunately, the universe of DSD functions is very restricting, as will be discussed further.

Optimality can also be easily checked for the class of functions called *read-polarity-once* (RPO) Boolean functions (CALLEGARO ET AL, 2012), where each polarity (positive or

negative) of a variable appears at most once in the minimal factored expression, *e.g.*  $h=(!a*d+c)*(a+b)$ . Notice that the function  $h$  it is not RO, since variable ‘ $a$ ’ appears twice, and  $h$  is also not a DSD function, since sub-functions  $g1=(!a*d+c)$  and  $g2=(a+b)$  is not disjoint (variable ‘ $a$ ’ appears in the support of  $g1$  and  $g2$ ). Similar to the DSD class of functions, the RPO class is also a superset of RO functions.

This section presents a comparison between RO, DSD and RPO classes over the occurrence of these classes in MCNC circuits (IWLS, 2005). Results confirm that the number of RPO functions are quite broader than RO as well as DSD functions. This means that there is room for improvement in factoring algorithms, enhancing the final quality of digital circuits.

Besides the comparison, this section presents a new method to synthesize RPO functions (CALLEGARO ET AL, 2013). The concept of the proposed algorithm is simpler than (CALLEGARO ET AL, 2012) and is able to synthesize RPO functions in shorter runtime. The proposed algorithm was able to efficiently find optimal solutions with up to 16 literals, while other methods cannot (YOSHIDA; FUJITA, 2011) (MARTINS ET AL, 2012).

## 6.1 Boolean function representation

A Boolean function can be represented in a two-level representation, like sum-of-products (SOP) or product-of-sums (POS) form. For example, the exclusive-or (XOR) operation can be represented in SOP form as  $f=(a!*b+!a*b)$  and in a POS form as  $f=(a+b)(!a+!b)$ . Two-level minimization was extensively studied by several authors, targeting programmable logic arrays (PLA) *MINI* (HONG; CAIN; OSTAPKO, 1974), *ESPRESSO* (BRAYTON ET AL, 1984), *Presto-II* (BARTHOLOMEUS; MAN, 1985), *PALMINI* (NGUYEN; PERKOWSKI; GOLDSTEIN, 1987), *ESPRESSO-SIGNATURE* (MCGEER; SANGHAVI; BRAYTON; VINCENTELLI, 1993), *Scherzo* (COUDERT, 1994), and *BOOM* (HLAVICKA; FISER, 2001).

The factored form (or multi-level representation) is the most widespread way to represent Boolean functions. In this representation, the main goal is to express a given Boolean function with the minimal literal count as possible. For example, let  $f=(a!b!d!e+!abde+!abc!d+!b!cde)$  be a given Boolean function to be factored. Different approaches can result in completely different solutions as shown in Table 13. Algebraic factorization (SENTOVICH ET AL, 1992) algorithms are scalable and run in a very short

time. However, results are not good in most cases. Similarly, graph-based approaches (MINTZ, GOLUMBIC, 2005) can run in a short runtime and keep the solution very near to optimal, whereas Boolean methods (MARTINS ET AL, 2012) can run in a feasible runtime just for few (e.g. 5) input variables, reaching optimal solutions in most cases.

Table 13 – Comparison between different factoring approaches.

Method	Factored form	Literals
Algebraic	$f = !a(!cde + (!d + e)bc) + !b(!cde + !d!ea)$	15
Graph-based	$f = !e!d!ba + (e + !d)(c + d)(b!a + !c!b)$	12
Boolean	$f = (!a + !b)(!d + e)(!cd + !ea + bc)$	10

At this point, a question could be raised:

*“How could we know how far are the factoring algorithms from reaching optimal (minimal literal count) solutions?”*

In order to answer this question, we can simply build an algorithm that always reaches the optimal solution, no matter how much time is consumed. However, in order to check if a simple 5-input function is in optimal form, such algorithm can take more than days to give the optimal solution, making our strategy unfeasible. Indeed, the problem of finding a minimal literal count expression for general Boolean functions belongs to the NP-hard class of problems. Nevertheless, optimality can be easily checked and guaranteed for special classes of functions, including read-once, disjoint support decomposition, and read-polarity-once Boolean functions.

## 6.2 Definition and properties of read-polarity-once functions

**Definition 15:** A Boolean function is called *read-polarity-once* (RPO) if each polarity (positive or negative) of a variable appears at most once in the minimum factored form (CALLEGARO ET AL, 2012).

**Lemma 5:** A positive (negative) unate variable contributes with at least one positive (negative) literal in a factored form.

**Lemma 6:** A binate variable contributes with at least two literals (one positive and one negative) in a factored form.

**Theorem 10:** A function represented by an RPO expression can be proved as minimum literal form if each unate variable contributes exactly one literal and each binate variable contributes exactly two literals (one positive and one negative).

**Proof:** straightforward by lemmas 5 and 6.

**Example:** Let  $f$  be a Boolean function defined  $f = !abd + ac + bc$ . The variable ‘ $a$ ’ is binate in  $f$ , while variables  $\{b, c, d\}$  are positive unate. In this sense, the minimal factored form for representing the given function is  $f = (!a * d + c) * (a + b)$ . The circuit that represents this RPO function is shown in Figure 30. Notice the binate variable ‘ $a$ ’ appears twice (once as positive and once as negative literal), whereas the unate variables  $\{b, c, d\}$  appear only once. This means that this function is an RPO function, and can be proved minimal as stated in Theorem 10.

It is worth mentioning that, as shown in Table 6, for the universe of functions with up to 5 inputs and 6 inputs, the number of RPO functions is two and three orders of magnitude larger than the number of DSD functions, respectively. For this reason, a method that is able to factorize RPO functions into minimal literal count forms is highly desired.

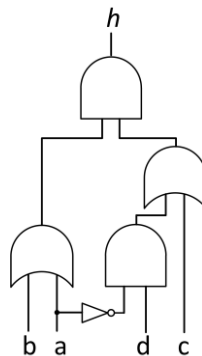


Figure 30 – Example a read-polarity-once function  $f = (!a * d + c) * (a + b)$ .

### 6.3 Proposed method for synthesis of RPO functions

This section presents a method to synthesize minimal literal count forms for RPO functions. The method is based on the concept of positive and negative transition sets possible for each variable. The definition of positive and negative transition sets is presented in section 6.3.1. The method is able to detect if two literals must be grouped through an AND or OR logic operation by computing transition sets. The intuition for this is presented in section

6.3.2, while the strict tests for bonding are presented in section 6.3.3. The complete algorithm uses a literal intersection graph, and it is described in section 6.3.4.

### 6.3.1 Positive and negative transitions of a variable

Let  $f(X)$  be a Boolean function defined over the variable set  $X = \{x_0, \dots, x_{n-1}\}$ .

**Definition 16:** Let  $x_i \in X$ . The *positive transition* (PT) set w.r.t.  $x_i$  in function  $f$  is defined as:

$$PT(f, x_i) = !f(x_i = 0) * f(x_i = 1) \quad (63)$$

**Definition 17:** Let  $x_i \in X$ . The *negative transition* (NT) set w.r.t.  $x_i$  in function  $f$  is defined as:

$$NT(f, x_i) = f(x_i = 0) * !f(x_i = 1) \quad (64)$$

The set of positive/ negative transitions of a variable can be interpreted as containing all possible input vectors that can be set in order to make an input transition in  $x_i$  visible in the output of the function  $f$ . This information is the basis of our RPO algorithm.

**Transition example:** Let  $f = (!a b d + a c + b c)$  be a Boolean function. If we want to observe the PT w.r.t. variable ‘ $a$ ’ appearing in the output of  $f$ , we must set the inputs ‘ $b$ ’ and ‘ $c$ ’ to 0 and 1, respectively. The resulting function is presented in Equation (67). However, if we want to observe the NT of ‘ $a$ ’, we must set the variables ‘ $b$ ’, ‘ $c$ ’ and ‘ $d$ ’ to 1, 0, 1, respectively. The negative transition function is presented in Equation (68). The reader can confirm this information by looking the circuit depicted in Figure 30. It is important to notice that in this step we are in the Boolean domain, meaning that the circuit does not exist and is cited here just for explaining the definition of transition sets.

$$f(a = 0) = b c + b d \quad (65)$$

$$f(a = 1) = c \quad (66)$$

$$PT(f, a) = !b c \quad (67)$$

$$NT(f, a) = b !c d \quad (68)$$

### 6.3.2 Intuition for grouping variables by transition test

The transition information is of great importance in the proposed method. In order to explain why, we ask for the reader to look at the Figure 30 again and try to figure it out how

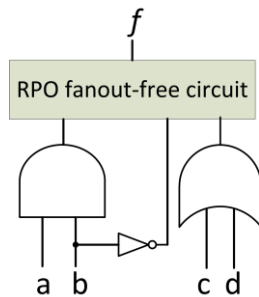
to solve the next question: “How could we cancel the positive transition w.r.t. variable ‘a’ using the smallest input assignments as possible?” The answer to this question is apparent in Equation (67). In order to cancel the PT of ‘a’, we can just set the input variable ‘b’ to 1, since ‘b’ will control the OR output (Figure 30 – left), turning the positive polarity ‘a’ irrelevant. There is another way to cancel the PT of ‘a’, which consist of setting the input variable ‘c’ to 0. This will force the AND gate (Figure 30 – top) to evaluate “b!ad”, which does not depend on positive variable ‘a’ anymore.

The transition set analysis gives us valuable information of a relationship between variables. We already have the information that the assignment of variables ‘b’ and ‘c’ could cancel the PT of ‘a’. But the opposite is not true for both ‘b’ and ‘c’ variables. In Equation (69) it is possible to see that if we assign ‘a’ to 1, we cancel the PT of ‘b’. The same is not true for the PT of ‘c’ in Equation (70), since there is no way to cancel the PT of ‘c’ by just assigning the variable ‘a’ (i.e. input vector “b!d” does not depend on ‘a’ and enable the PT of ‘c’).

$$PT(f, b) = !a c + !a d \quad (69)$$

$$PT(f, c) = a + b !d \quad (70)$$

When a transition cancellation is mutual between two variables, we can say that these variables can be grouped through an AND or an OR gate. However, we are not able yet to identify the correct gate to group them. In order to proper group these variables, consider the RPO circuit in Figure 31. Let  $f$  be a Boolean function over the variable set  $\{a, b, c, d\}$ . Let the PT for each variable be defined as Equations.(71)-(74). It is possible to observe a pattern between variables ‘a’ and ‘b’ as well as between variables ‘c’ and ‘d’. This leads to the working principle of literal grouping, presented in the next section.



$$PT(f, a) = b \dots \quad (71)$$

$$PT(f, b) = a \dots \quad (72)$$

$$PT(f, c) = !d \dots \quad (73)$$

$$PT(f, d) = !c \dots \quad (74)$$

Figure 31 – Example of an RPO function.

### 6.3.3 Literals and grouping definition

Let  $f(X)$  be a Boolean function defined over the variable set  $X = \{x_0, \dots, x_{n-1}\}$ . We need to split each input variable  $x_i$  into positive (“ $x_i$ ”) and negative (“ $!x_i$ ”) literals, in order to compose the literal set  $L$ . Each member of  $L$  will be defined as a triple (*expression, function, transition*), which will be simply accessed as  $x_i.exp$ ,  $x_i.func$  and  $x_i.trans$ , respectively. This means that each positive literal  $x_i$  will lead to the triple (“ $x_i$ ”,  $x_i$ ,  $PT(f, x_i)$ ). The same process will be applied to the negative literals, leading to the triple (“ $!x_i$ ”,  $!x_i$ ,  $NT(f, x_i)$ ). It is important to notice that each member of  $L$  could not have a transition function equals to the constant 0. This means that this specific element is irrelevant to compose the given function  $f$ , e.g. if  $x_i$  has a positive unate behavior in  $f$ , the negative literal “ $!x_i$ ” must not appear in the minimum factored form of  $f$ .

From this point on, we can define a Boolean function  $f$  over the set of literals  $L$ . In this way, we define possible grouping as follows:

**Definition 18 (AND grouping):** Let  $\{y_i, y_j\} \in L$  and  $(y_i \neq y_j)$ . Two literals  $y_i$  and  $y_j$  can be grouped into an AND gate if the Equation (75) is satisfied.

$$(y_i.trans * !y_j.func \equiv 0) \wedge p(y_j.trans * !y_i.func \equiv 0) \quad (75)$$

**Definition 19 (OR grouping):** Let  $\{y_i, y_j\} \in L$  and  $(y_i \neq y_j)$ . Two literals  $y_i$  and  $y_j$  can be grouped into an OR gate if the Equation (76) is satisfied.

$$(y_i.trans * y_j.func \equiv 0) \wedge (y_j.trans * y_i.func \equiv 0) \quad (76)$$

### 6.3.4 Literal cluster intersection graph

From this point, it is possible to present our main data structure. Let  $f$  be a Boolean function over the set of literals  $L$ . We define the literal cluster intersection (LCI) graph  $G = (V, E)$ , where each vertex in  $V$  is a triple (*expression, function, transition*) (initially,  $V = L$ ) and each edge in  $E$  contains a bit flag representing an AND / OR operator. There is an AND (OR) edge between  $v_i$  and  $v_j$ , where  $\{v_i, v_j\} \in V$ , if they can be grouped through the formula Equation (75)-(76).

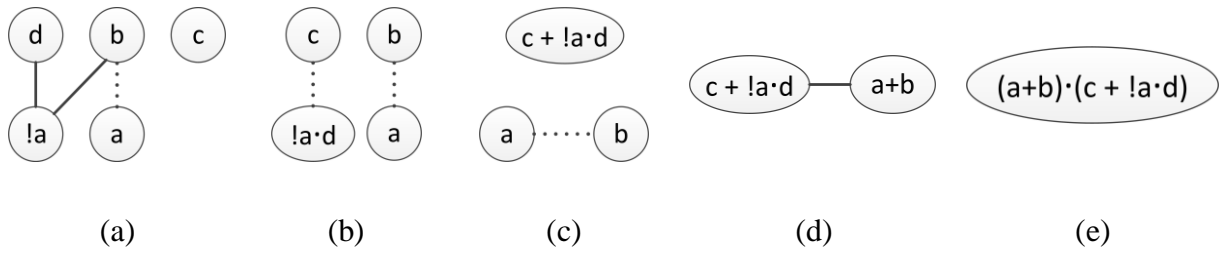


Figure 32 – Step by step example of the proposed factoring algorithm for RPO functions. The initial graph (a) contains the relationship between literals, where solid edges represent AND grouping while dashed edges represent OR grouping. The algorithm proceeds and chooses the solid edge between “ $!a$ ” and “ $d$ ”, leading to the graph shown in (b). The algorithm continues with (c) and (d) steps, reaching the optimal solution in (e).

The LCI graph  $G$  for the function in  $f=(!abd+ac+bc)$  is depicted in Fig. 4(a). The algorithm then chooses an arbitrary edge  $e$  between  $v_i$  and  $v_j$ . The vertices  $v_i$  and  $v_j$  are then removed from the graph  $G$ . A new vertex  $v_k$  is created, and the triple  $(expression, function, transition)$  is properly filled with the AND / OR operations for the  $v_k.expression$  and  $v_k.function$  fields, while  $v_k.transition=(v_i.transition+v_j.transition)$ , independent of the edge type.

A full example is depicted in Figure 32. Notice that in this example a “good” ordering was chosen. If the first edge selected in Figure 32-(a) had been the one between “ $!a$ ” and “ $b$ ”, an RPO solution would not have been found. If such decision occurs, the algorithm must try a backtracking strategy until a solution is obtained. If no solution is found, the given Boolean function does not belong to the RPO class of functions.

## 6.4 Decomposing non-RPO functions

We presented an algorithm that find, whenever possible, a RPO realization for a given Boolean function. Unfortunately, there are functions that cannot be represented by an RPO form. In this section we show how one can decompose such functions and, hopefully, find simpler RPO functions.

Let  $f(X)$  be a non-RPO function. We can decompose  $f$  into simpler functions and then test if such functions are RPO. One example of decomposition is the *Shannon expansion* (also known as Shannon decomposition and Boole's expansion). The Shannon expansion of  $f(X)$  w.r.t a variable  $x_i \in X$  is defined as follows (SHANNON, 1949):



$$f(X) = !x_i \cdot f(x_i = 0) + x_i \cdot f(x_i = 1) \quad (77)$$

The process of decomposing a non-RPO function into simpler RPO functions is called herein 1-step Shannon decomposition. Notice that in this case both positive and negative cofactors of  $f$  w.r.t a variable  $x_i$  must be RPO functions.

**Example:** Let  $f = ab + ac + bc$  (a non-RPO function). We can try to decompose it by applying a decomposition strategy. Since  $f(a = 0) = bc$  and  $f(a = 1) = b + c$  are both RPO function, then we can represent  $f$  by a 1-step Shannon decomposition w.r.t. variable “ $a$ ”, resulting  $f = \bar{a}(bc) + a(b + c)$ .

All 2-input functions are RPO, as shown in Table 6. Out of 256 functions of 3-inputs, 230 are RPO. All the remaining 26 functions can be decomposed by applying 1-step Shannon decomposition, since the all resulting cofactors are functions depending on at most 2-inputs. For the universe of 65,536 functions with up to 4-inputs, 20,748 are RPO (31.7%) and 42,496 (64.8%) can be synthesized by applying 1-step Shannon decompositions.

We can also apply another decomposition approach when the 1-step Shannon expansion fails. The positive (negative) Davio decomposition can be obtained from the Boolean difference and negative (positive) cofactor functions. The Boolean difference is the XOR of negative and positive cofactors w.r.t a variable  $x_i$  as shown in Eq. (78). The positive and negative Davio expansions are presented in Eq. (79) and (80), respectively.

$$\frac{\partial f}{\partial x_i} = f(x_i = 0) \oplus f(x_i = 1) \quad (78)$$

$$f(X) = f(x_i = 0) \oplus x_i \cdot \frac{\partial f}{\partial x_i} \quad (79)$$

$$f(X) = f(x_i = 1) \oplus !x_i \cdot \frac{\partial f}{\partial x_i} \quad (80)$$

**Example:** Let  $f = !a!b!c!d + !a!b!c!d + !a!b!c!d + !a!b!c!d + a!b!c!d$ . Since  $f$  is a non-RPO function we can try the 1-step Shannon decomposition. However, none of the support variables of  $f$  have both positive and negative cofactors as RPO functions, i.e. there is no 1-step Shannon decomposition for  $f$ . On the other hand, one can obtain a 1-step Davio decomposition since  $f(b = 1) = !a * (!d * !c + d * c)$  and  $\frac{\partial f}{\partial b} = !a + !d * !c$ , resulting  $f = !a (!d !c + d c) \oplus !b (!a + !d !c)$ .

### 6.4.1 Proposed decompositions

Both Shannon, negative and positive Davio decompositions use two subfunctions in order to reconstruct  $f$ . In (BECKER; DRECHSLER, 1995), it is proved that these three decompositions suffice when representing  $f$  into two subfunctions. In the following, two decompositions that use three subfunctions are presented.

Let  $f(X)$  be a Boolean function defined over the variable set  $X = \{x_1, \dots, x_n\}$  and  $x_i \in X$ . Let  $f_{x_i=0}$  ( $f_{x_i=1}$ ) represent the negative (positive) cofactor of  $x_i$  w.r.t  $f$ . The Universal Quantification of function  $f$  w.r.t a variable  $x_i$  is defined as a AND of  $x_i$ 's negative and positive cofactors:  $\forall x_i f = f_{x_i=0} \cdot f_{x_i=1}$ . The Existential Quantification of  $f$  w.r.t a variable  $x_i$  is the OR of  $x_i$ 's cofactors:  $\exists x_i f = f_{x_i=0} + f_{x_i=1}$ . The negative derivative of  $f$  w.r.t a variable  $x_i$  is defined  $\frac{\partial f}{\partial x_i^-} = f_{x_i=0} \cdot \overline{f_{x_i=1}}$  while the positive derivative of  $f$  w.r.t a variable  $x_i$  is defined  $\frac{\partial f}{\partial x_i^+} = \overline{f_{x_i=0}} \cdot f_{x_i=1}$ . Table 14 summarizes these definitions and notation.

Table 14 – Summary of components used in the proposed decomposition. The negative (positive) cofactor w.r.t  $x_i$  is represented by  $f_{x_i=0}$  ( $f_{x_i=1}$ ).

Name	Notation	Definition
Existential Quantification	$\exists x_i f$	$f_{x_i=0} + f_{x_i=1}$
Negative Derivative	$\frac{\partial f}{\partial x_i^-}$	$f_{x_i=0} \cdot \overline{f_{x_i=1}}$
Positive Derivative	$\frac{\partial f}{\partial x_i^+}$	$\overline{f_{x_i=0}} \cdot f_{x_i=1}$
Universal Quantification	$\forall x_i f$	$f_{x_i=0} \cdot f_{x_i=1}$

Two quantified decomposition are presented. These decompositions use three subfunctions (tri-decomposition) instead of two subfunctions. Eq. (81) presents the Universal-based decomposition that uses negative and positive derivatives as well as the Universal Quantification.

$$f(X) = \overline{x_i} \cdot \frac{\partial f}{\partial x_i^-} + x_i \cdot \frac{\partial f}{\partial x_i^+} + \forall x_i f \quad (81)$$

**Example:** Let  $f = \overline{b}\overline{c}\overline{d}\overline{e} + \overline{a}\overline{b}\overline{c}d + \overline{a}\overline{c}\overline{d}\overline{e} + \overline{a}\overline{b}cde$  be a non-RPO function. There is no 1-step Shannon decomposition neither a 1-step Davio decomposition for such function.

However, we can apply the Universal-based decomposition, shown in Eq. (81), to search for 1-step RPO decompositions since  $\frac{\partial f}{\partial e^-} = (\bar{a} + \bar{b})\bar{c}\bar{d}$ ,  $\frac{\partial f}{\partial e^+} = \bar{a}\bar{b}c\bar{d}$  and  $\forall x_i f = \bar{a}\bar{b}c\bar{d}$  resulting  $f = \bar{e}(\bar{a} + \bar{b})\bar{c}\bar{d} + e\bar{a}\bar{b}c\bar{d} + \bar{a}\bar{b}c\bar{d}$ .

A decomposition that uses Existential Quantification, negative and positive derivatives called Existential-based decomposition is shown in Eq. (82).

$$f(X) = \left( \bar{x}_i + \frac{\overline{\partial f}}{\partial x_i^-} \right) \cdot \left( x_i + \frac{\overline{\partial f}}{\partial x_i^+} \right) \cdot \exists x_i f \quad (82)$$

**Example:** Let  $f = \bar{b}\bar{c}\bar{d} + \bar{a}\bar{c}\bar{d} + \bar{a}\bar{b}\bar{e} + \bar{a}\bar{d}\bar{e} + \bar{a}\bar{b}e + \bar{a}\bar{c}\bar{e}$  be a non-RPO function. There is no 1-step Shannon decomposition, neither a 1-step Davio decomposition nor a 1-step Universal-based decomposition for such function. However, we can apply the Existential-based decomposition, shown in Eq. (82), to search for 1-step RPO decompositions since  $\frac{\partial f}{\partial e^-} = \bar{a}\bar{b}(c + d)$ ,  $\frac{\partial f}{\partial e^+} = \bar{a}\bar{b}c\bar{d}$  and  $\exists x_i f = \bar{a} + \bar{b}\bar{c}\bar{d}$  resulting  $f = (\bar{e} + \bar{a}\bar{b}(c + d)) \cdot (e + \bar{a}\bar{b}c\bar{d}) \cdot (\bar{a} + \bar{b}\bar{c}\bar{d})$ .

## 6.5 Experimental results

The first experiment was carried out over the set of all 5-input NPN-class (negation-permutation-negation) functions, grouped in 616,125 Boolean classes by equivalence through input permutation, negation of its inputs and/or negation of the output. Instead of running the algorithm for all 5-input Boolean space ( $2^{32}$  functions), the NPN-class benchmark was chosen. This benchmark can represent the functionality of all Boolean space of five variables in a more compact set, without losing generality. The platform used to perform these results was a Linux system with Intel Core i5 2400 processor and 2GB main memory.

To run our algorithm for all the 616,125 functions, 44 seconds of execution time were needed. The worst case optimization was 1ms and the average case was less than 70  $\mu$ s. The in-house implementation of the X-Factor algorithm (MINTZ, GOLUMBIC, 2005) took 50 minutes to complete, resulting in total 12,530,011 literals. The FC algorithm (MARTINS ET AL, 2012) resulted in 11,292,029 literals in total and took 6 hours to complete. It is important to notice that both algorithms are designed to factorize general Boolean functions, while the RPO algorithm does not synthesize non-RPO functions.

Comparative results evaluating the efficiency of the proposed algorithm are shown in Table 15, considering the set of 1,462 RPO functions extracted from 5-input NPN-class benchmark. Our algorithm presented better results in terms of both runtime and literal count when compared to Quick Factor (QF) (SENTOVICH ET AL, 1992), Good Factor (GF) (SENTOVICH ET AL, 1992), Functional Composition (FC) algorithm (MARTINS ET AL, 2012) and X-Factor (MINTZ, GOLUMBIC, 2005) factoring algorithms.

Table 15 – Total number of literals and runtime obtained when factoring 1,462 RPO functions using different approaches.

	<b>QF</b>	<b>GF</b>	<b>FC</b>	<b>X-Factor</b>	<b>RPO 2013</b>	<b>This work</b>
<b>Literals</b>	16,086	15,671	13,754	13,253	13,064	13,064
<b>Runtime</b>	1.9s	2.3s	21s	7.1s	5.7s	0.7s

In the second experiment, we performed a study over the MCNC circuits (IWLS 2005). We have extracted functions up to 10 inputs from the benchmarks through the k-cuts method (CHATTERJEE; MISHCHENKO; BRAYTON, 2006), (MACHADO ET AL, 2012). Unfortunately, the FC and X-Factor algorithms were too slow to complete. Our algorithm took 36s, 2m36s and 6m53s to factorize the functions extracted from the circuits through k-cuts with up to 6, 8 and 10 inputs, respectively. As it is possible to see in Table 16, RPO functions are still quite frequent in benchmark circuits, meaning that there is room for improvement in factoring algorithms, enhancing the final quality of digital circuits.

The last experiment consists on the analysis of non-RPO functions over a set of benchmark functions: all functions up to three and four inputs; a set of 5-input functions grouped by input negation / permutation and output negation (NPN); a set of all disjoint-support-decomposable (DSD) functions with up to 6-inputs and a set of full-DSD, partial-DSD and non-DSD that appeared in (HUANG ET AL, 2013). This study is to evaluate how frequent can non-RPO functions be decomposed using a 1-step RPO expansion. The results are shown in Table 17. The number of functions that have 1-dist Shannon, Davio or Quantifier-Based expansions are shown in columns Shannon, Davio and Quantifier-Based, respectively. Column 1-dist RPO show the number of functions that have at least one Shannon, Davio or Quantifier-Based 1-step decomposition. Finally, column RPO + 1-dist RPO represent the number of functions that are RPO or have 1-dist decomposition.

As expected, all functions with up to 3-inputs are RPO or can be represented by a 1-step RPO expansion. Interestingly, 99% of the functions with up to 4-inputs are RPO or are 1-step RPO. For the NPN set of 5-input functions and the set of all DSD functions with up to 6-input this number is 64% and 98%, respectively. For the benchmarks of full-, partial-, and non-DSD functions ranging from 6- to 16-inputs, it is shown that 1-step RPO are quite frequent as well.

## **6.6 Conclusions**

This section discussed the relevance of read-once (RO), disjoint support decomposition (DSD) and read-polarity-once (RPO) classes of functions. For the universe of functions with up to 5-inputs, the class of RPO functions is two orders of magnitude larger than RO and DSD ones. The study of the MCNC benchmark confirmed this claim. Besides the study of classes, an efficient method that guarantees minimal factored forms for the class of RPO functions was also proposed. Results show the efficiency of our method which is able to find better results in shorter runtime when compared to state-of-the-art factoring algorithms.

Table 16 – Decomposition of circuits into k-cuts, with k=6, k=8 and k=10. The number of read-once (RO), disjoint support decomposition (DSD) and read-polarity-once (RPO) functions is presented.

Benchmark	K = 6					K = 8					K = 10				
	Total	RO	DSD	RPO	Time (ms)	Total	RO	DSD	RPO	Time (ms)	Total	RO	DSD	RPO	Time (ms)
9symml	41	12	16	31	389	9	1	2	2	182	1	0	0	0	46
alu2	109	65	75	90	165	39	10	17	19	749	6	1	3	3	5718
alu4	194	103	125	148	131	115	40	60	64	12632	48	6	12	15	5349
apex6	157	44	64	153	51	144	30	50	139	54	117	22	41	110	75
apex7	55	35	47	52	16	45	24	36	42	16	42	22	34	39	21
b1	4	2	3	4	0	4	2	3	4	0	4	2	3	4	1
c8	28	5	11	28	6	22	5	5	22	6	20	3	3	20	8
cc	21	13	13	21	5	20	12	12	20	5	20	12	12	20	5
cht	36	0	0	36	7	36	0	0	36	7	36	0	0	36	9
cm150a	6	1	1	1	1	5	0	0	0	1	5	0	0	0	2
cm151a	4	0	0	2	1	4	0	0	2	0	3	0	0	0	1
cm152a	3	0	0	1	0	2	0	0	0	0	2	0	0	0	0
cm162a	10	6	10	10	3	7	3	6	6	12	6	2	2	6	7
cm163a	7	3	3	7	2	6	2	2	6	3	5	1	1	5	3
cm42a	10	10	10	10	2	10	10	10	10	2	10	10	10	10	2
cm82a	3	0	1	0	1	3	0	1	0	0	3	0	1	0	1
cm85a	6	0	2	2	2	6	0	2	2	2	3	0	1	1	2
cmb	11	8	8	11	3	8	5	5	8	3	7	5	5	7	191
comp	22	8	15	15	6	11	3	7	7	6	11	3	7	7	6
cordic	9	5	7	7	3	8	5	6	6	4	5	0	1	1	7
count	24	8	8	24	6	21	4	5	21	8	20	3	5	20	12
cu	16	14	14	16	2	13	8	8	13	3	12	8	8	11	3
dalu	235	36	75	134	81	186	20	45	63	386	151	16	16	44	18569
decod	16	16	16	16	3	16	16	16	16	3	16	16	16	16	3
des	1035	309	534	931	1892	778	58	305	563	2242	350	6	29	206	239
example2	94	62	67	89	11	79	25	49	48	352	72	30	35	60	3037
f51m	21	7	9	11	3	8	1	3	3	2	8	1	3	3	2
frg1	26	21	21	21	4	24	16	16	17	6	17	8	8	9	6
i10	582	345	432	544	548	486	265	340	432	1719	464	230	273	371	29981
i1	18	16	17	18	2	17	15	16	17	2	17	15	16	17	1
i2	66	65	65	66	7	34	32	32	34	8	28	26	26	26	10
i3	42	42	42	42	6	22	22	22	22	5	22	22	22	22	6
i4	62	62	62	62	10	38	38	38	38	10	34	34	34	34	12
i5	76	76	76	76	11	66	66	66	66	12	68	68	68	68	18
i6	67	0	0	1	11	67	0	0	1	11	67	0	0	1	13
i7	67	1	3	3	9	67	1	3	3	10	67	1	3	3	10
i8	284	123	123	284	42	170	42	42	168	40	242	78	78	187	109
i9	229	171	171	227	39	75	5	5	10	29	68	0	0	1	44
k2	543	525	526	536	77	486	434	439	474	31055	374	243	246	315	264658
lal	25	15	25	25	5	23	14	23	23	5	21	12	21	21	7
majority	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
my_adder	25	0	8	1	7	23	0	6	1	8	21	0	4	1	12
pair	338	199	242	316	150	252	115	152	208	3153	239	110	147	187	2510
parity	5	0	5	0	1	3	0	3	1	1	2	0	2	0	1
pcler8	24	17	23	24	3	22	15	18	22	4	21	14	18	21	4
pcler8	13	6	13	13	3	12	4	5	12	3	11	4	11	11	3
pm1	16	14	15	16	2	14	12	13	14	2	13	11	12	13	2
sct	21	13	18	21	4	18	8	15	18	3	17	7	14	17	5
tcon	16	8	8	16	1	16	8	8	16	0	16	8	8	16	0
term1	42	23	29	34	10	34	18	23	26	56	20	6	11	12	24370
ttt2	35	15	22	31	23	24	4	11	17	4277	23	3	10	16	138
unreg	16	0	0	16	2	16	0	0	16	3	16	0	0	16	2
vda	283	249	252	272	32612	197	120	120	136	98615	131	47	48	72	57509
x1	89	79	79	85	10	74	63	63	68	12	61	45	45	54	20
x2	14	12	13	13	1	10	7	8	8	1	7	4	5	5	1
x3	167	57	78	167	21	138	29	50	135	24	111	17	37	107	30
x4	109	89	96	108	13	83	41	48	82	16	78	55	62	77	23
z4ml	6	0	2	1	0	4	0	1	0	1	4	0	1	0	1
<b>Total</b>	<b>5484</b>	<b>3015</b>	<b>3600</b>	<b>4889</b>	<b>36426</b>	<b>4121</b>	<b>1678</b>	<b>2241</b>	<b>3207</b>	<b>155771</b>	<b>3264</b>	<b>1237</b>	<b>1478</b>	<b>2344</b>	<b>412825</b>
<b>Ratio</b>	<b>1</b>	<b>0.55</b>	<b>0.66</b>	<b>0.89</b>	<b>-</b>	<b>1</b>	<b>0.41</b>	<b>0.54</b>	<b>0.78</b>	<b>-</b>	<b>1</b>	<b>0.38</b>	<b>0.45</b>	<b>0.72</b>	<b>-</b>

Table 17 – Results on the analysis of non-RPO functions over a set of benchmark functions. Column RPO show the number of RPO functions. The number of functions that have 1-dist Shannon, Davio or Quantifier-Based expansions are shown in columns Shannon, Davio and Quantifier-Based, respectively. Column 1-dist RPO shows the number of functions that have at least one of the previous decompositions. Finally, column RPO + 1-dist RPO represent the number of functions that are RPO or have 1-dist decomposition.

Benchmark	Functions	RPO (%)	Shannon (%)	Davio (%)	Quantifier-Based (%)	1-dist RPO (%)	RPO + 1-dist RPO (%)
All 3-in functions	256	230 (89.8%)	26 (10.2%)	24 (9.4%)	26 (10.2%)	26 (10.2%)	256 (100.0%)
All 4-in functions	65,536	20,750 (31.7%)	42,496 (64.8%)	43,720 (66.7%)	42,688 (65.1%)	44224 (67.5%)	64,974 (99.1%)
NPN 5-in	616,125	1,421 (0.2%)	177,507 (28.8%)	282,288 (45.8%)	317,378 (51.5%)	398307 (64.6%)	399,727 (64.9%)
All DSD up to 6-in	2,311,640	1,699,626 (73.5%)	544,680 (23.6%)	574,640 (24.9%)	585,288 (25.3%)	585480 (25.3%)	2,285,108 (98.9%)
Full-DSD 6-in	1,000,000	992,754 (99.3%)	4,614 (0.5%)	4,788 (0.5%)	4,788 (0.5%)	4788 (0.5%)	997,542 (99.8%)
Full-DSD 8-in	1,000,000	925,190 (92.5%)	5,267 (0.5%)	5,735 (0.6%)	5,716 (0.6%)	5740 (0.6%)	930,930 (93.1%)
Full-DSD 10-in	100,000	98,117 (98.1%)	968 (1.0%)	1,054 (1.1%)	1,043 (1.0%)	1054 (1.1%)	99,171 (99.2%)
Full-DSD 12-in	100,000	97,496 (97.5%)	1,605 (1.6%)	1,612 (1.6%)	1,614 (1.6%)	1614 (1.6%)	99,110 (99.1%)
Full-DSD 14-in	10,000	9,104 (91.0%)	114 (1.1%)	126 (1.3%)	125 (1.3%)	126 (1.3%)	9,230 (92.3%)
Full-DSD 16-in	10,000	9,355 (93.6%)	65 (0.7%)	118 (1.2%)	118 (1.2%)	118 (1.2%)	9,473 (94.7%)
Partial-DSD 6-in	1,000,000	823,030 (82.3%)	165,327 (16.5%)	167,389 (16.7%)	168,430 (16.8%)	168950 (16.9%)	991,980 (99.2%)
Partial-DSD 8-in	1,000,000	683,462 (68.3%)	262,012 (26.2%)	283,453 (28.3%)	291,517 (29.2%)	294676 (29.5%)	978,138 (97.8%)
Partial-DSD 10-in	100,000	65,562 (65.6%)	32,017 (32.0%)	30,682 (30.7%)	32,500 (32.5%)	32949 (32.9%)	98,511 (98.5%)
Partial-DSD 12-in	100,000	50,818 (50.8%)	40,838 (40.8%)	39,327 (39.3%)	42,329 (42.3%)	44061 (44.1%)	94,879 (94.9%)
Partial-DSD 14-in	10,000	6,861 (68.6%)	2,001 (20.0%)	2,189 (21.9%)	2,251 (22.5%)	2297 (23.0%)	9,158 (91.6%)
Partial-DSD 16-in	10,000	4,653 (46.5%)	2,900 (29.0%)	3,148 (31.5%)	3,451 (34.5%)	3519 (35.2%)	8,172 (81.7%)
Non-DSD 6-in	1,000,000	134,275 (13.4%)	809,998 (81.0%)	815,630 (81.6%)	827,986 (82.8%)	834525 (83.5%)	968,800 (96.9%)
Non-DSD 8-in	1,000,000	27,280 (2.7%)	534,261 (53.4%)	536,794 (53.7%)	590,349 (59.0%)	663394 (66.3%)	690,674 (69.1%)
Non-DSD 10-in	100,000	83 (0.1%)	26,722 (26.7%)	23,068 (23.1%)	26,671 (26.7%)	32540 (32.5%)	32,623 (32.6%)
Non-DSD 12-in	100,000	364 (0.4%)	12,456 (12.5%)	13,432 (13.4%)	14,924 (14.9%)	17220 (17.2%)	17,584 (17.6%)
Non-DSD 14-in	10,000	22 (0.2%)	940 (9.4%)	900 (9.0%)	1,073 (10.7%)	1263 (12.6%)	1,285 (12.9%)
Non-DSD 16-in	10,000	20 (0.2%)	350 (3.5%)	390 (3.9%)	380 (3.8%)	460 (4.6%)	480 (4.8%)
Total	9,653,557	5,650,474 (58.5%)	2,667,164 (27.6%)	2,842,366 (29.4%)	2,960,645 (30.7%)	3,137,331 (32.5%)	8,787,805 (91.0%)

## 7 CONCLUSIONS

The problem of factoring and decomposition for Boolean functions is  $\Sigma_2^P$ -complete for general functions. Efficient and exact algorithms can be created for existing class of functions known as read-once (RO), disjoint-support decomposable (DSD) and read-polarity-once (RPO) functions.

This dissertation discussed logic synthesis methods focusing on the realization of specific classes of Boolean functions. Four new algorithms for synthesis of Boolean functions were presented. The first contribution was a synthesis method that finds a read-once realization for a target function. The method was designed based on a divide-and-conquer strategy. Given a BDD with  $m$  nodes and  $n$  inputs, the **RO\_BY\_COFACTOR** has a worst-case performance of  $O(mn)$ . This method was implemented in logic synthesis framework named SwitchCraft (CALLEGARO ET AL, 2010).

The second and third contributions were algorithms for synthesis of DSD functions (CALLEGARO ET AL, 2015). A top-down approach checks if there is an OR, AND, or XOR decomposition based on sum-of-products, product-of-sums and exclusive-sum-of-products inputs, respectively. This method is also available in the SwitchCraft tool. The another DSD method runs in a bottom-up fashion and is based on Boolean difference and cofactor analysis. The proposed method needs  $O(n \cdot \log n)$  cofactor and  $O(n)$  equivalence test operations to perform AND and XOR decomposition. The algorithm was implemented into both ABC and SwitchCraft frameworks (CALLEGARO ET AL, 2015b).

The last contribution is a new method to synthesize RPO functions (CALLEGARO ET AL, 2013). The method is based on the concept of positive and negative transition sets possible for each variable. The method is able to detect if two literals must be grouped through an AND or OR logic operation by computing transition sets. This method is available in both SwitchCraft and ABC tool (command *test\_rpo*).





## REFERENCES

- AKERS, S. B. J. **On a theory of Boolean functions.** Journal of the Society for Industrial & Applied Mathematics, no. 7.4, p. 487-498. 1959.
- ALPERT, C. J.; MEHTA, D. P.; SAPATNEKAR, S. S. **Handbook of algorithms for physical design automation.** [S.l.]: CRC press, 2008.
- ANGLUIN, D.; HELLERSTEIN, L.; KARPINSKI, M.; **Learning read-once formulas with queries.** J. ACM 40 185–210. 1993.
- ASHENHURST, R. **The decomposition of switching functions.** Proc. of the Int'l Symp. on the Theory of Switching, pages 74–116, Apr. 1957.
- BARTHOLOMEUS M.; MAN H. D. **Presto-II: Yet another logic minimizer for programmed logic arrays.** Proc. Int. Symp. Circ. Syst. 1985.
- BECKER, B., DRECHSLER, R. **How many decomposition types do we need?** In Proceedings of the 1995 European conference on Design and Test. IEEE Computer Society, Washington, DC, USA, 438-. 1995.
- Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification.
- BERTACCO, V.; DAMIANI, M. **Disjunctive decomposition of logic functions.** in Proc. of Int'l Conf. on Computer-Aided Design (ICCAD), p. 78-82. 1997.
- BERTACCO, V.; OLUKOTUN, K. **Efficient state representation for symbolic simulation.** In DAC, Proceedings of Design Automation Conference, June 2002.
- BOROS, E.; GURVICH, V.; HAMMER, P.L. **Read-once decompositions of positive boolean functions.** RUTCOR Research Report RRR 01/24-94. p. 254–283. 1994.
- BOROS, E.; IBARAKI, T.; MAKINO, K. **Error-free and best-fit extensions of partially defined Boolean functions.** Inform. and Comput. 140. 1998

- BRAYTON, R. K. **Factoring logic functions**. IBM Journal of Research and Development, vol. 31, no. 2, p.187-98. Mar 1987.
- BRAYTON, R. K.; SANGIOVANNI-VINCENTELLI, A. L.; MCMULLEN C. T; HACHTEL G. D., **Logic Minimization Algorithms for VLSI Synthesis**. Kluwer Academic Publishers, Norwell, MA, USA. 1984.
- BUCHFUEHRER, D.; UMANS, C. **The complexity of Boolean formula minimization**. Journal of Computer and System Sciences 77: 142. 2011.
- BSHOUTY, N.; HANCOCK, T.R.; HELLERSTEIN, L. **Learning boolean read-once formulas with arbitrary symmetric and constant fan-in gates**. J. Comput. System Sci. 50 521–542. 1995
- BUTLER, J. T. **On the number of functions realized by cascades and disjunctive networks**, IEEE Trans. Computers, C-24), 681-690. 1975
- CALLEGARO, V. Disponível em: <http://www.inf.ufrgs.br/~vcallegaro/functions/>. Acesso em: 30 Junho 2016.
- CALLEGARO, V.; MARRANGHELLO, F. S. ; MARTINS, M. G. A. ; RIBAS, R. P. ; REIS, A. I. **Bottom-up disjoint-support decomposition based on cofactor and boolean difference analysis**. In: 33rd IEEE International Conference on Computer Design (ICCD), 2015, New York City. 2015 33rd IEEE International Conference on Computer Design (ICCD). p. 680. 2015
- CALLEGARO, V.; MARTINS, M. G. A. ; RIBAS, R. P. ; REIS, A. I. **DSD Synthesis Based on Variable Intersection Graphs**. In: South Symposium on Microelectronics (SIM), 2015.
- CALLEGARO, V.; MARTINS, M. G. A. ; RIBAS, R. P. ; REIS, A. I. . **A Domain-Transformation Approach to Synthesize Read-Polarity-Once Boolean Functions**. JICS. Journal of Integrated Circuits and Systems (Ed. Português), v. 9, p. 60-69, 2014.
- CALLEGARO, V.; MARTINS, M. G. A. ; RIBAS, R. P. ; REIS, A. I. **Read-polarity-once Boolean functions revisited**. In: International Workshop on Logic & Synthesis (IWLS), 2013, Austin. Proceedings of International Workshop on Logic & Synthesis. 2013.
- CALLEGARO, V.; MARTINS, M. G. A.; RIBAS, R. P.; REIS, A. I. **Read-Polarity-Once Functions**. In: International Workshop on Logic and Synthesis (IWLS'2012). Berkeley, CA, USA. 2012.

CALLEGARO, V; MARQUES, F. S.; KLOCK, C. E.; DA ROSA JUNIOR, L. S.; RIBAS, R. P.; REIS, A. I. **SwitchCraft: a framework for transistor network design**. In Proceedings of the 23rd symposium on Integrated circuits and system design (SBCCI '10). São Paulo, SP, Brazil. 2010.

CANTEAUT, A.; VIDEAU, M.; **Symmetric Boolean functions**. IEEE Transactions on Information Theory, vol. 51, no. 8, p. 2791-2811, Aug. 2005.

CHATTERJEE, S.; MISHCHENKO, A.; BRAYTON, R. **Factor Cuts**. IEEE/ACM International Conference on Computer Aided Design, San Jose, CA, p. 143-150. 2006.

CHRZANOWSKA-JESKE, M. **Regular symmetric arrays for non-symmetric functions**. Proceedings of the 1999 IEEE International Symposium on. Vol. 1. IEEE, 1999.

COOK, S. A. **The complexity of theorem-proving procedures**. In Proceedings of the third annual ACM symposium on Theory of computing (STOC '71). ACM, New York, NY, USA, 151-158. 1971

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L; STEIN, C. **Data structures for Disjoint Sets**, Introduction to Algorithms (Second ed.), Chapter 21, MIT Press, p. 498–524. 2001.

COUDERT O. **Two-Level Logic Minimization: An Overview** .Integration, 17-2, p. 97-140, Oct. 1994.

CURTIS, H. A. **A new approach to the design of switching circuits**. D. Van Nostrand Co, Princeton, NJ, 1962.

DA ROSA, L. S.; MARQUES, F. S.; SCHNEIDER, F.; RIBAS, R. P.; REIS, A. I. **A comparative study of CMOS gates with minimum transistor stacks**. Proc. of the 20th Annual Conf. on Integrated Circuits and Systems Design, (SBCCI), p. 93-98. 2007.

DAVIO, M.; DESCHAMPS, J. P.; THAYSE, A. **Discrete and Switching Functions**, McGraw-Hill. 1978.

ESPRESSO-BOOK-EXAMPLES.

<http://embedded.eecs.berkeley.edu/pubs/downloads/espresso>. (visited: June, 2016).

GOLUMBIC, M. C.. **Algorithmic Graph Theory and Perfect Graphs**. second ed., Academic Press, NewYork, 1980, Ann. Discrete Math. 57. 2004.

GOLUMBIC, M. C.; MINTZ, A. **Factoring logic functions using graph partitioning**. In Proc. of the IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD), p.195-199. 1999.

GRAY, F. **Pulse Code Communication**, US patent #2,632,058. 1953.

HACHTEL, G. D.; SOMENZI, F.. **Logic Synthesis and Verification Algorithms**, Springer, 564p. 2006.

HALMOS, P. R. **Naive Set Theory**. Princeton, N.J.: Van Nostrand. 1960

HAYES, J. P. **A Nand Model for Fault Diagnosis in Combinational Logic Networks**. IEEE Transactions on Computers, vol. C-20, no. 12, p. 1496-1506, Dec. 1971.

HAYES, J. P. **Enumeration of fanout-free Boolean functions**, J. ACM, 23, 700-709. 1976.

HAYES, J. P. **The fanout structure of switching functions**. J. ACM, vol. 22, no. 4. p. 551-71. 1975

HLAVICKA J.; FISER P. **BOOM-a heuristic Boolean minimizer**. Proc. Of Int. Conf. on Computer-Aided Design (ICCAD), p.439-442. 2001.

HONG, S. J.; CAIN R. G.; OSTAPKO D. L. **MINI: A heuristic approach for logic minimization**. IBM J., vol. 18, p.443 -458 1974.

HOPCROFT, J.; TARJAN, R. **Efficient algorithms for graph manipulation**. Communications of the ACM 16 (6): 372–378. 1973.

HUANG, Z.; WANG L.; NASIKOVSKIY, Y.; MISHCHENKO, A. **Fast Boolean matching based on NPN classification**. in Proc. of Int'l Conf. on Field Programmable Technology (ICFPT), 2013.

IWLS 2005 Benchmarks: <http://www.iwls.org>.

KAGARIS, D; HANIOTAKIS, T. **A methodology for transistor-efficient supergate design**. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 15, no. 4, p. 488-492, Apr. 2007.

KANAGAL, K.; LI, JIAN; AND DESHPANDE, AMOL. **Sensitivity analysis and explanations for robust query evaluation in probabilistic databases**. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11). ACM, New York, NY, USA, 841-85, 2011.

KARCHMER, M.; LINIAL, N.; NEWMAN, I.; SAKS, M.; WIGDERSON, A.. **Combinatorial characterization of read-once formulae**. Discrete Math. 114. p. 275–282. 1993.

KARNAUGH, M. **The map method for synthesis of combinational logic circuits**. Trans. AIEE. pt. I, 72(9):593–599, 1953.

KARPLUS, K. **Using if-then-else dags to do technology mapping for field-programmable gate arrays**. Technical Report UCSC-CRL-90-43, Baskin Center for Computer Engineering & Information Sciences, 1990.

KODANDAPANI, K. L.; SETH, S. C. **On combinational networks with restricted fan-out**. IEEE Trans. Computers, 27 309-318. 1978

KOZLOWSKI, T. **Application of exclusive-OR logic in technology independent logic optimisation**. Ph.D. Thesis. January 1996.

KUTZSCHEBAUCH, T.; AND STOK, L. **Layout driven decomposition with congestion consideration**. In DATE, Design, Automation and Test in Europe Conference, pages 672–676, March 2002.

MACHADO, L., MARTINS, M., CALLEGARO, V., RIBAS R. P. AND REIS A. I., **KL-cut based digital circuit remapping**, NORCHIP, Copenhagen, 2012, p. 1-4. 2012.

MARTINS, M. G. A.; CALLEGARO, V; MACHADO, L.; RIBAS, R. P.; REIS, A. I.. **Functional Composition Paradigm and Applications**. International Workshop on Logic and Synthesis (IWLS'2012). Berkeley, CA. 2012.

MARTINS, M. G.A., CALLEGARO, V., RIBAS, R. P., REIES, A. I. **Efficient method to compute minimum decision chains of Boolean functions**. In Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI (GLSVLSI '11). ACM, New York, NY, USA, 419-422. 2011.

MARTINS, M. G. A.; DA ROSA JR., L. S.; RASMUSSEN, A.; RIBAS, R. P.; REIS, A. I. **Boolean factoring with multi-objective goals**. Proc. of Int'l Conf. on Computer Design (ICCD), p. 229-234. 2010.

MATSUNAGA, Y. **An Efficient Algorithm finding simple disjoint decompositions using BDDs**. IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences, no. 85, vol. 12, p.2715 - 2724 , 2002.

- MCCLUSKEY, E.J., JR. **Minimization of Boolean Functions**. Bell System Technical Journal 35 (6). Nov 1956.
- MCGEER P.; SANGHAVI J.; BRAYTON R.; SANGIOVANNI-VINCENTELLI, A. L. **ESPRESSO-SIGNATURE: A New Exact Minimizer for Logic Functions**. Proc. Of Design Automation Conference (DAC), p. 618-624. 1993.
- MICHELI, G. D. **Synthesis and Optimization of Digital Circuits**. [S.l.]: McGraw-Hill Higher Education, 1994.
- MINATO, S.; DE MICHELI, G. **Finding all simple disjunctive decompositions using irredundant sum-of-products forms**. in Proc. of of Int'l Conf. on Computer-Aided Design (ICCAD), p. 111-117. 1998.
- MINTZ, A.; GOLUMBIC, M. C. **Factoring Boolean functions using graph partitioning**. Discrete Applied Mathematics, 2005.
- MISHCHENKO A.; CHATTERJEE, S.; BRAYTON, R. **DAG-aware AIG rewriting a fresh look at combinational logic synthesis**. In Proceedings of the 43rd annual Design Automation Conference. ACM, New York, NY, USA, 532-535. 2006.
- MISHCHENKO A.; PERKOWSKI M. **Fast Heuristic Minimization of Exclusive Sum-of-Products**. Submitted to the 5th International Reed-Muller Workshop, August 2001.
- MISHCHENKO, A. BRAYTON, R. **Faster logic manipulation for large designs**. in Proc. of Int'l Workshop on Logic Synthesis (IWLS), 2013.
- MISHCHENKO, A. **Enumeration of irredundant circuit structures**. in Proc. of Int'l Workshop on Logic Synthesis (IWLS), 2014.
- MISHCHENKO, A.; BRAYTON, R. **Faster Logic Manipulation for Large Designs**. International Workshop on Logic and Synthesis (IWLS). 2013
- MISHCHENKO, A.; CHATTERJEE, S.; JIANG, R.; BRAYTON, R. **FRAIGs: A Unifying Representation for Logic Synthesis and Verification**. Tech. rep., EECS Dept. UC Berkeley, 2005.
- MISHCHENKO, A.; EEN, N.; BRAYTON, R. CASE, M.; CHAUHAN, P.; SHARMA, N. **A semi-canonical form for sequential AIGs**, Proc. Design, Automation and Test in Europe (DATE'13), p. 797-802. 2013.

MISHCHENKO, A.; STEINBACH, B.; PERKOWSKI, M. **An algorithm for bi-decomposition of logic functions.** In Proceedings of the 38th annual Design Automation Conference. 2001.

MURGAI, R.; NISHIZAKI, Y; SHENOY, N. V.; BRAYTON, R. K.; AND SANGIOVANNI-VINCENTELLI, A. **Logic synthesis for programmable gate arrays.** In DAC, Proceedings of Design Automation Conference, pages 620–625, June 1990.

NGUYEN L. B.; PERKOWSKI M. A.; GOLDSTEIN N. B. **PALMINI-Fast Boolean Minimizer for Personal Computer.** Design Automation Conference, p. 615-621. 1987.

PEER, J.; PINTER, R.. **Minimal decomposition of Boolean functions using non-repeating literal trees.** In Proc. of the IFIP Workshop on Logic and Architecture Synthesis, IFIP TC10 WD10.5, p.129-39. 1995.

PERKOWSKI, M. A.; GRYGIEL, S. **A Survey of Literature on Function Decomposition Version IV.** Portland State University. 1995.

PLAZA, S.; BERTACCO, V. **STACCATO Disjoint Support Decompositions from BDDs through Symbolic Kernels,** Asia and South Pacific Design Automation Conference (ASP-DAC). 2005.

POSSANI, V. N.; CALLEGARO, V.; REIS, A. I.; RIBAS, R. P.; MARQUES, F. S.; DA ROSA, L. S. **Graph-Based Transistor Network Generation Method for Supergate Design.** IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 24, no. 2, p. 692-705, Feb. 2016.

POSSANI, V. N.; SOUZA, R. S.; DOMINGUES, J. S.; AGOSTINI, L. V.; MARQUES, F. S.; DA ROSA JR, L. S. **Optimizing transistor networks using a graph-based technique.** Journal of Analog Integrated Circuits and Signal Processing, vol. 73, no. 3, p. 841-850, Dec. 2012.

QUINE, W. V. **A Way to Simplify Truth Functions.** The American Mathematical Monthly 62 (9): 627–631. Nov, 1955.

QUINE, W. V. **The Problem of Simplifying Truth Functions.** The American Mathematical Monthly 59 (8): 521–531. Oct, 1952.

ROTH, J.; KARP, R. **Minimization over Boolean graphs.** *IBM Journal*, p. 227-238, Apr. 1962.



RUDELL, R. **Dynamic variable ordering for ordered binary decision diagrams.** Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design. IEEE Computer Society Press, 1993.

SASAO T. **EXMIN2: a simplification algorithm for exclusive-OR-sum-of-products expressions for multiple-valued-input two-valued-output functions.** IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol.12, no.5, p.621,632, May 1993.

SASAO, T. **DECOMPOS: An Integrated System for Functional Decomposition.** International Workshop on Logic and Synthesis. 1998

SASAO, T. **Multiple-valued decomposition of generalized Boolean functions and the complexity of programmable logic arrays.** IEEE Transactions on Computers, C-30(9):635–643, September 1981.

SASAO, T. **Switching Theory for Logic Synthesis.** Springer, 1999.

SASAO, T.; BUTLER, J. **On bi-decomposition of logic functions.** Int'l Workshop of Logic & Synthesis (IWLS). 1997.

SASAO, T; BUTLER, J. T. **Worst and best irredundant sum-of-products expressions.** IEEE Transactions on Computers, vol. 50, no. 9, p. 935-948, Sep 2001.

SEN, P.; DESHPANDE, A; GETOOR, L. **Read Once Functions and Query Evaluation in Probabilistic Databases.** Proceedings of the VLDB Endowment. 2010

SENTOVICH E.; SINGH K.; LAVAGNO L.; MOON, C.; MURGAI R.; SALDANHA, A.; SAVOJ, H.; STEPHAN, P.; BRAYTON R.; SANGIOVANNI-VINCENTELLI, A. L. **SIS: A system for sequential circuit synthesis.** Tech. Rep. UCB/ERL M92/41. UC Berkeley, Berkeley. 1992.

SHANNON, C. E. **A Symbolic Analysis of Relay and Switching Circuits.** Trans. AIEE 57 (12): 713–723. 1938.

SHANNON, C. **The synthesis of two-terminal switching circuits.** Bell System Technical Journal, Wiley Online Library, v. 28, n. 1, p. 59–98, 1949.

SLOANE, N. J. A.; PLOUFFE, S. **The Encyclopedia of Integer Sequences,** Academic Press, 1995.

STANION, T.; SECHEN, C. **Boolean division and factorization using binary decision diagrams**. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 9, p.1179-84. Sep. 1994.

VENN, J. **On the diagrammatic and mechanical representation of propositions and reasonings**, Philosophical Magazine Series 5, Vol. 10, Iss. 59, 1880

YOSHIDA H.; FUJITA M. **Exact minimum factoring of incompletely specified logic functions via quantified Boolean satisfiability**. IPSJ Trans. on System LSI Design Methodology, vol. 4, Feb. 2011.

YOSHIDA, H.; IKEDA, M.; ASADA, K. **Exact minimum logic factoring via quantified Boolean satisfiability**. In Proc. of IEEE Int'l Conf. on Electronics, Circuits and Systems (ICECS), p.1065-68. 2006.

YOSHIDA, H; FUJITA, M.. **Exact minimum factoring of incompletely specified logic functions via quantified Boolean satisfiability**, IPSJ Trans. on System LSI Design Methodology, vol. 4, p.70-79. 2011.

ZHANG, J. S.; MISHCHENKO, A; BRAYTON, R; CHRZANOWSKA-JESKE, M. **Symmetry detection for large Boolean functions using circuit representation, simulation, and satisfiability**. In Proceedings of the 43rd annual Design Automation Conference. 2006.

ZHU, J.; ABD-EL-BARR, M. **On the optimization of MOS circuits**. IEEE Trans. on Circuits and Systems I: Fundamentals, Theory and Applications, vol. 40, no. 6, p. 412-422, June 1993.

## APÊNDICE <RESUMO DA TESE>

# MINIMIZAÇÃO ÓTIMA DE CLASSES ESPECIAIS DE FUNÇÕES BOOLEANAS

### 1. INTRODUÇÃO

O fluxo de projeto de síntese de circuitos é normalmente dividido em três etapas: síntese arquitetural, síntese lógica e de síntese física. A síntese arquitetural, muitas vezes chamado de síntese de alto nível, consiste em transformar uma descrição algorítmica do comportamento desejado do circuito em um formato de hardware, como o formato de transferência de dados entre os registradores – RTL (*Register Transfer Level*). Normalmente, essas descrições algorítmicas são representadas em um formato similar a linguagem C (por exemplo, *System C*) ou em uma linguagem de descrição comportamental de hardware – HDL (*Hardware Description Language*), como por exemplo, VHDL ou Verilog.

O processo de síntese lógica tem sido uma das áreas de maior sucesso comercial no campo de automação de projetos eletrônicos – EDA (*Electronic Design Automation*). A grande maioria dos dispositivos digitais que usamos no nosso dia-a-dia foi concebida por um conjunto de ferramentas de síntese lógica. A tarefa de síntese lógica consiste em várias etapas. Estas etapas podem ser diferentes de acordo com a natureza do circuito, por exemplo sequencial ou combinacional. O principal objetivo da síntese lógica é determinar a estrutura primitiva de um circuito, ou seja, a sua representação em nível de portas padrão. A síntese lógica é geralmente dividida em três fases: otimizações independentes de tecnologia, mapeamento tecnológico e otimizações dependentes de tecnologia (MICHELI, 1994). A primeira aplica transformações que não dependem da tecnologia em si, mas do comportamento funcional de uma rede Booleana, e.g. algoritmos de fatoração e de decomposição. A fase de mapeamento tecnológico faz o casamento entre partes do circuito para células de uma biblioteca, que tem informações sobre a tecnologia alvo. Por fim, na fase dependente de tecnologia, se aplicam otimizações no circuito mapeado que levam fortemente em consideração informações da tecnologia, por exemplo, redimensionamento de células e duplicação de lógica. A síntese física, ou síntese no nível geométrico, consiste principalmente

de duas tarefas: o posicionamento de blocos lógicos, que distribui fisicamente as células e o roteamento, que conecta os sinais através de fios (ALPERT; Mehta; SAPATNEKAR, 2008).

Este trabalho visa a síntese de funções booleanas no âmbito de um fluxo de projeto de circuitos digitais, mais precisamente na fase de síntese lógica. O escopo deste trabalho pode ser considerado mais amplo, visto que os algoritmos propostos para síntese de funções Booleanas podem ter aplicação em diferentes áreas que não sejam síntese de circuitos, por exemplo, inteligência artificial (ANGLUIN; HELLERSTEIN; KARPINSKI, 1993; BSHOUTY; HANCOCK; HELLERSTEIN, 1995) e bancos de dados (SEN; DESHPANDE; GETOOR, 2010) (KANAGAL; LI; DESHPANDE, 2011).

### **Motivação e desafios**

O processo de fatorar funções Booleanas é uma operação fundamental na síntese lógica (BRAYTON, 1987; HACHTEL; SOMENZI, 2006). Fatoração é o processo de derivar uma equação algébrica, ou forma fatorada, representando uma dada função Booleana. Por exemplo,  $F = x_1x_2 + x_1x_3x_4 + x_1x_3x_5$  pode ser fatorada em uma equação logicamente equivalente  $F = x_1(x_2 + x_3(x_4 + x_5))$ .

Qualquer função lógica pode ser representada por expressões fatoradas distintas. A tarefa de fatorar funções Booleanas em fórmulas mais compactas e equivalentes é uma das operações básicas dos estágios iniciais da síntese lógica (HACHTEL; SOMENZI, 2006). Na maioria dos estilos de projeto, como portas lógicas CMOS, o número de dispositivos necessários para a realização de uma função Booleana corresponde quase que diretamente com sua forma fatorada em termos de contagem de literais. Gerar uma forma fatorada ótima, i.e. a equação com menor número de literais, é um problema  $\Sigma_2^P$ -complete (GOLUMBIC; MINTZ, 1999). Logo, algoritmos heurísticos foram desenvolvidos de forma a se obter boas soluções fatoradas (BRAYTON, 1987; STANION; SECHEN, 1994; MINTZ; GOLUMBIC, 2005; HACHTEL; SOMENZI, 2006; YOSHIDA; FUJITA, 2011). Algumas heurísticas bem conhecidas incluem X-Factor (MINTZ; GOLUMBIC, 2005), que provê boas soluções, mas não garante equações mínimas. Em (LAWLER, 1964), o autor afirma prover a fatoração exata. Porém, o método de Lawler não é escalável e é impraticável mesmo para algumas funções com apenas quatro entradas. Recentemente, novas abordagens têm melhorado o processo de fatoração visando soluções exatas, mas a escalabilidade e tempo de execução continuam a ser o principal gargalo (YOSHIDA; IKEDA; ASADA, 2006; YOSHIDA; FUJITA, 2011; MARTINS ET AL., 2012).

Como fatoração e decomposição para funções genéricas é um problema  $\Sigma_2^P$ -complete, uma boa estratégia é a identificação de funções Booleanas que são mais fáceis de serem sintetizadas. Este é o caso das classes de funções Booleanas *read-once*, decomposição disjunta de suporte e *read-polarity-once*.

Uma forma fatorada é chamada de *read-once* (RO) se cada variável aparece uma única vez. Uma função Booleana é RO se ela puder ser representada por uma forma RO (HAYES, 1975). Por exemplo, a função representada por  $F = x_1x_2 + x_1x_3x_4 + x_1x_3x_5$  é RO, pois pode ser fatorada em  $F = x_1(x_2 + x_3(x_4 + x_5))$ .

Uma função Booleana  $f(X)$  pode ser decomposta usando funções mais simples  $g$  and  $h$ , de forma que  $f(X) = h(g(X_1), X_2)$  sendo  $X_1, X_2 \neq \emptyset$ , e  $X_1 \cup X_2 = X$  (ASHENHURST, 1957), (CURTIS, 1962). Uma decomposição disjunta de suporte (DSD - *disjoint-support decomposition*) é um caso especial de decomposição funcional, onde o conjunto de entradas  $X_1$  e  $X_2$  não compartilham elementos, *i.e.*,  $X_1 \cap X_2 = \emptyset$ . Grosso modo, funções DSD podem ser representadas por uma expressão *read-once* onde o operador ou-exclusivo ( $\oplus$ ) também pode ser utilizado como operador básico. Por exemplo,  $F = x_1(x_2 \oplus (x_4 + x_5))$ .

Uma forma *read-polarity-once* (RPO) é uma forma fatorada onde cada polaridade (positiva ou negativa) de uma variável aparece no máximo uma única vez. Uma função Booleana é RPO se ela puder ser representada por uma forma fatorada RPO (CALLEGARO ET AL, 2012). Por exemplo, a função  $F = \bar{x}_1x_2x_4 + x_1x_3 + x_2x_3$  é RPO, pois pode ser fatorada em  $F = (\bar{x}_1x_4 + x_3)(x_1 + x_2)$ .

A motivação para a pesquisa destas classes especiais de funções é que, além de mais simples para síntese, estas classes são de interesse especial no contexto de projeto de circuitos digitais, visto que a ocorrência das mesmas é extremamente frequente em circuitos (PEER; PINTER, 1995), (MISHCHENKO, BRAYTON, 2013) (CALLEGARO ET AL, 2014). O desafio é, portanto, a criação de métodos eficientes e exatos que possam sintetizar tais classes.

## Objetivo

Esta tese apresenta quatro novos algoritmos para a síntese de funções Booleanas. A primeira contribuição é um método de síntese que encontra uma realização *read-once* para uma dada função Booleana. O método é baseado em uma estratégia de divisão-e-conquista. A solução *read-once* para uma função alvo é obtida através da obtenção de soluções *read-once* para sub-funções mais simples: cofatores negativos e positivos (fase da divisão). Estas soluções mais simples são então compostas (fase de conquista), resultando em uma solução *read-once* para o problema original (função alvo). O método não depende especificamente de uma estrutura de dados, pois necessita apenas das operações de cofatoração e checagem de tautologia e antilogia (testa se a função é uma constante 1 ou 0, respectivamente).

A segunda e terceira contribuições são métodos para síntese de funções DSD. (CALLEGARO ET AL, 2015a). Uma abordagem *top-down* checa se existe uma decomposição OR, AND ou XOR baseado em entradas de soma-de-produtos, produto-de-somas e soma-exclusiva-de-produtos, respectivamente. O outro método é uma abordagem *bottom-up* e é baseado na análise de diferenças Booleanas e cofatores (CALLEGARO ET AL, 2015b). Dois testes simples resultam em condições necessárias e suficientes para identificar decomposições AND e XOR.

A última contribuição é um novo método para síntese de funções RPO (CALLEGARO ET AL, 2013). O método é baseado no conceito de conjunto de transições positivas e negativas das variáveis de entrada. O método é capaz de detectar se dois literais devem ser agrupados utilizando uma operação lógica AND ou OR pela análise dos conjuntos de transições.

## 2. MÉTODO PROPOSTO PARA SÍNTESE DE FUNÇÕES READ-ONCE

Um método de síntese que encontra, sempre que possível, uma realização *read-once* (RO) para uma função alvo é apresentado. O método se baseia no fato de que a classe de função RO é fechada para operações de cofator, isto é, o cofator de uma função RO é uma função RO (HAYES, 1975). Dada uma função  $f(X)$  dependendo de  $n$  entradas e uma variável  $x_i \in X$ , a ideia é obter recursivamente árvores *read-once* para os cofatores negativos e positivos de uma variável  $x_i$  e, baseado nestas duas árvores, decidir o ponto de inserção de  $x_i$  em uma destas árvores.

O método foi implementado utilizando como estrutura de dados básica um diagrama de decisão binária, reduzido e ordenado (ROBDD – *Reduced and Ordered Binary Decision Diagram*). Note que o método não depende especificamente de um diagrama de decisão binária (BDD – *Binary Decision Diagram*). Porém, BDDs são candidatos naturais, pois os cofatores das variáveis já estão convenientemente calculados. A plataforma usada para avaliar o método foi um sistema Linux com processador Intel Core i5 2400 e 4 GB de memória principal.

O primeiro experimento consiste na análise isolada do método de conquista (**RO\_COMPOSITION**). O método recebe duas árvores *read-once* em formato canônico e, baseado nesta informação, compõe se possível uma árvore *read-once* para a função alvo. A Figura 1 mostra que o método **RO\_COMPOSITION** roda em tempo linear em relação ao número de entradas.

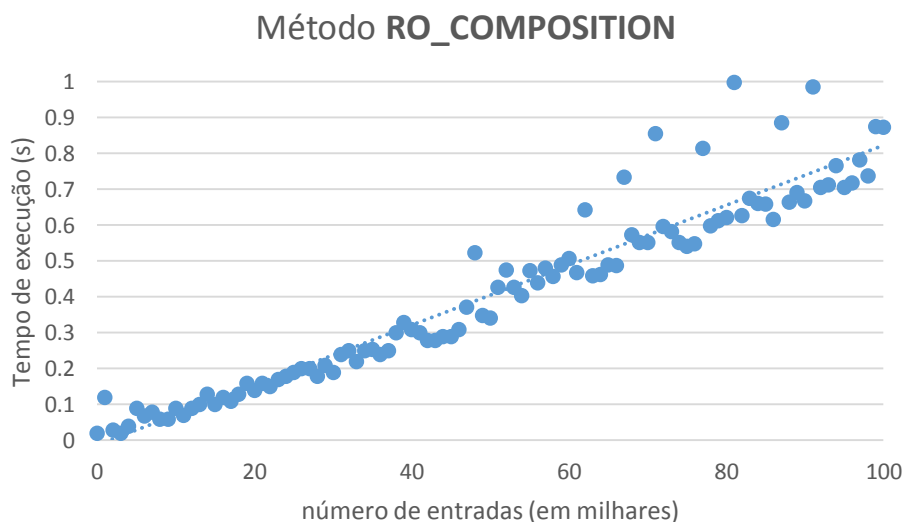


Figura 1 – Análise de tempo de execução do método **RO\_COMPOSITION**. Os resultados mostram que o método roda em  $O(n)$ .

O segundo experimento consiste na obtenção de ROBDDs para funções *read-once*. A ordem das variáveis escolhidas para a criação dos ROBDDs foi completamente aleatória. Note que os ROBDDs gerados poderiam ter um número menor de nodos se utilizado um ordenamento dinâmico de variáveis (RUDELL, 1993). Entretanto, nossa ideia é mostrar como o método se comporta mesmo quando o número de nodos do BDD é bastante grande. Resultados obtidos após a execução do método **RO\_BY\_COFACTOR** nestes BDDs são exibidos na Tabela 1. Como foi utilizada uma ordem randômica de variáveis, o maior BDD, representando uma função com 51 entradas, tinha 637,185 nodos e levou 8.76 segundos para ser executado. Em todos os casos, soluções *read-once* foram encontradas para cada nodo do BDD. No total, 1,709,418 nodos de BDD foram avaliados, levando 20.47 segundos para execução.

No terceiro experimento, o método foi testado para funções não-*read-once*. Como sabemos de antemão que o BDD não representa uma função *read-once*, nenhuma solução *read-once* deve ser encontrada para o nodo do topo do BDD. Porém, devem existir nodos do BDD que possam ser representados por expressões *read-once*; ao menos para nodos representando variáveis de entrada e nodos constantes.



Deixe uma função não-*read-once* (aleatoriamente gerada) ser representada por  $F_1 = \bar{x}_1 x_3 \bar{x}_4 x_5 x_6 + \bar{x}_2 x_3 x_4 \bar{x}_5 x_6 + x_1 \bar{x}_2 x_5 x_6 + x_2 \bar{x}_3 \bar{x}_5 \bar{x}_6 + x_1 x_3 x_5 x_6 + \bar{x}_1 x_2 x_3 x_4 x_5 + \bar{x}_1 \bar{x}_2 \bar{x}_4 x_6 + \bar{x}_1 \bar{x}_2 x_3 x_4 \bar{x}_5 + x_1 \bar{x}_3 \bar{x}_5 \bar{x}_6 + x_1 x_2 \bar{x}_3 \bar{x}_5 + x_2 x_3 \bar{x}_4 \bar{x}_5 x_6 + \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 + x_1 x_2 \bar{x}_4 x_5 \bar{x}_6 + x_1 x_2 x_4 \bar{x}_5 \bar{x}_6 + x_1 \bar{x}_3 x_4 x_5 + \bar{x}_1 x_2 x_3 \bar{x}_4 \bar{x}_6$ . O BDD resultante é mostrado na Figura 2. Nodos em verde representam funções *read-once*.

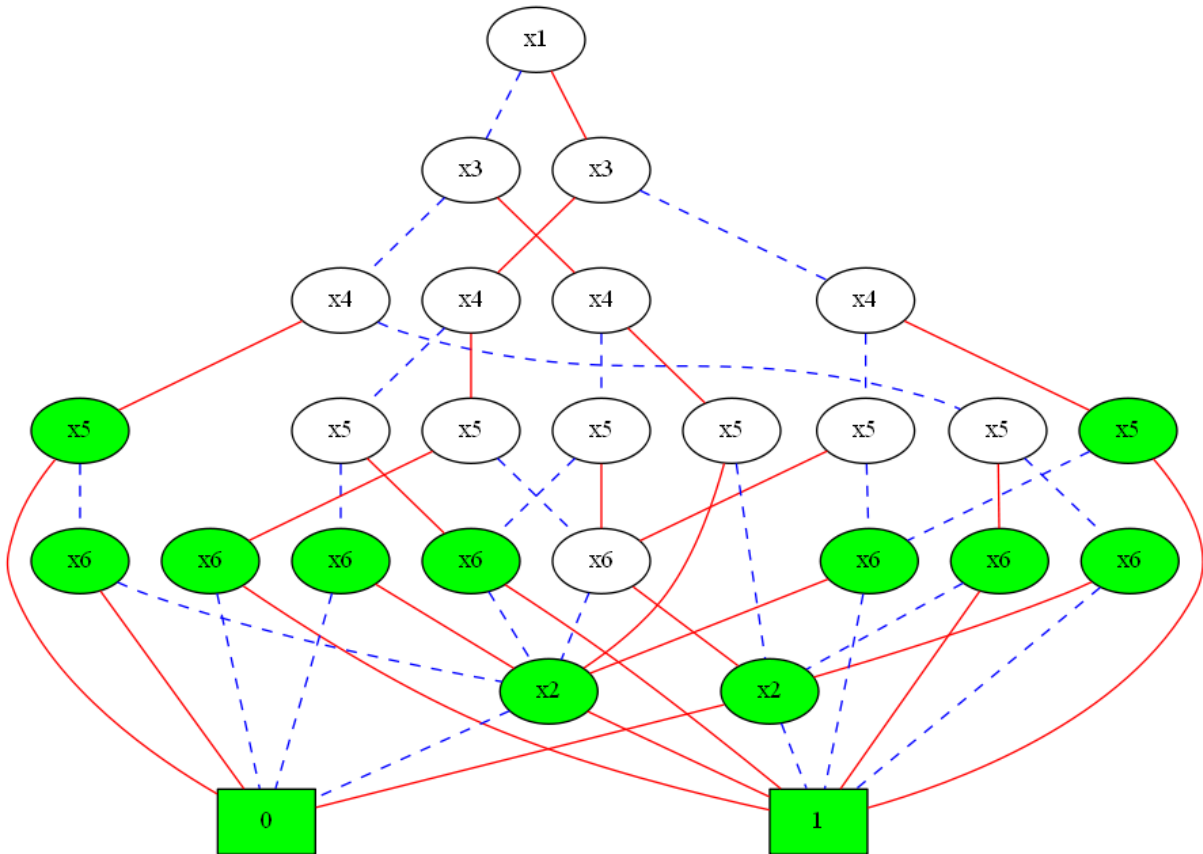


Figura 2 – ROBDD representando uma função não-*read-once*  $F_1 = \bar{x}_1 x_3 \bar{x}_4 x_5 x_6 + \bar{x}_2 x_3 x_4 \bar{x}_5 x_6 + x_1 \bar{x}_2 x_5 x_6 + x_2 \bar{x}_3 \bar{x}_5 \bar{x}_6 + x_1 x_3 x_5 x_6 + \bar{x}_1 x_2 x_3 x_4 x_5 + \bar{x}_1 \bar{x}_2 \bar{x}_4 x_6 + \bar{x}_1 \bar{x}_2 x_3 x_4 \bar{x}_5 + x_1 \bar{x}_3 \bar{x}_5 \bar{x}_6 + x_1 x_2 \bar{x}_3 \bar{x}_5 + x_2 x_3 \bar{x}_4 \bar{x}_5 x_6 + \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 + x_1 x_2 \bar{x}_4 x_5 \bar{x}_6 + x_1 x_2 x_4 \bar{x}_5 \bar{x}_6 + x_1 \bar{x}_3 x_4 x_5 + \bar{x}_1 x_2 x_3 \bar{x}_4 \bar{x}_6$ .

Nodos em verde representam funções *read-once*.

Tabela 1 – Tempo de execução do método **RO\_BY\_COFACTOR** sobre ROBDDs representando funções *read-once*. Os ROBDDs foram construídos utilizando uma ordem randômica de variáveis.

Entradas	#BDD	Execução (s)	Entradas	#BDD	Execução (s)
2	4	0.02	27	1,146	0.01
3	5	0.00	28	2,402	0.05
4	8	0.00	29	3,808	0.05
5	9	0.00	30	3,604	0.04
6	10	0.00	31	3,948	0.04
7	11	0.00	32	4,104	0.06
8	20	0.00	33	7,882	0.10
9	28	0.00	34	1,133	0.01
10	34	0.00	35	7,052	0.06
11	44	0.00	36	3,614	0.03
12	42	0.00	37	18,674	0.15
13	68	0.00	38	31,516	0.26
14	51	0.00	39	14,344	0.12
15	114	0.00	40	5,794	0.04
16	80	0.01	41	21,920	0.23
17	92	0.00	42	33,238	0.28
18	353	0.01	43	22,868	0.16
19	368	0.01	44	130,210	1.26
20	718	0.02	45	61,361	0.61
21	412	0.00	46	152,138	1.79
22	389	0.01	47	134,976	1.49
23	685	0.01	48	50,636	0.54
24	1,079	0.01	49	284,700	3.62
25	851	0.01	50	65,116	0.61
26	574	0.00	51	637,185	8.76

### 3. MÉTODO DE DECOMPOSIÇÃO TOP-DOWN PARA SÍNTESE DE FUNÇÕES DISJUNTAS DE SUPORTE

Esta seção apresenta um método para síntese de decomposição disjunta de suporte (DSD – *disjoint-support decomposition*) (CALLEGARO ET AL, 2015). O método proposto é baseado em um grafo de interseção de variáveis que é diretamente obtido dos termos Booleanos representando uma dada função  $f$  (GOLUMBIC, 2004), (MINTZ, GOLUMBIC, 2005). Caso este grafo seja desconectado em  $m$  componentes conectados, a função  $f$  pode ser decomposta em  $m$  subfunções  $h_0(X_0), \dots, h_{m-1}(X_{m-1})$ .

O operador de decomposição  $\circ \in \{\mathbf{AND}, \mathbf{OR}, \mathbf{XOR}\}$  é determinado de forma que a implementação DSD para  $f$  é obtido por  $f = \circ (h_0(X_0), \dots, h_{m-1}(X_{m-1}))$ , onde  $X_0, \dots, X_{m-1}$  são mutuamente disjuntos. O operador de decomposição  $\circ$  é um operador **OR** se os termos Booleanos foram obtidos dos cubos de uma soma-de-produtos irredundante (ISOP – *irredundant sum-of-products*) de  $f$ . O operador  $\circ$  é uma operação **AND** se os termos Booleanos são somas obtidas de um produto-de-somas irredundante (IPOS - *irredundant product-of-sums*). Finalmente,  $\circ$  é um operador **XOR** se os termos Booleanos vierem de produtos de uma soma-exclusiva-de-produtos (ESOP – *exclusive sum-of-products*).

#### Definições

Deixe  $F$  ser uma expressão em formato SOP, POS ou ESOP representando uma função Booleana  $f(X)$ . Um grafo de interseção de variáveis (VIG – *variable intersection graph*)  $G_F = (X, E)$  é um grafo não direcionado onde os vértices correspondem as variáveis em  $F$ , e existe uma aresta entre  $(x_i, x_j) \in E$ ,  $x_i, x_j \in X$  se e somente se  $x_i$  and  $x_j$  estão presente no mesmo termo Booleano (GOLUMBIC, 2004), (MINTZ, GOLUMBIC, 2005).

Deixe  $f = (a+b) \oplus (c \cdot d)$  ser representado por uma forma ISOP  $F = (!a!bcd + a!d + b!c + a!c + b!d)$  e por uma forma ESOP  $H = 1 \oplus (!a!b) \oplus (c \cdot d)$ . O VIG  $G_F$  e  $G_H$  são mostrados na Figura 3 (a) e Figura 3 (b), respectivamente.



Figura 3 – Um grafo de interseção de variáveis (VIG) obtido de uma (a) ISOP

$F = (!a!b \cdot c \cdot d + a!d + b!c + a!c + b!d)$  e (b) de uma forma ESOP  $H = 1 \oplus (!a!b) \oplus (c \cdot d)$ .

Na teoria dos grafos, um componente conectado de um grafo não direcionado é um subgrafo em que 1) qualquer dois vértices estão conectados entre si através de um caminho, e 2) não está conectado a nenhum outro vértice adicional no supergrafo (HOPCROFT, 1973). Por exemplo, o grafo  $G_F$  apresentado na Figura 3 (a) é conectado, *i.e.* contém exatamente um componente conectado  $\{a, b, c, d\}$ . O grafo  $G_H$ , exibido na Figura 3 (b), é desconectado e contém exatamente dois componentes conectados  $\{a, b\}$  e  $\{c, d\}$ .

De forma a exemplificar o método proposto, deixe uma função Booleana  $f$  ser representada pela seguinte expressão:

$$\begin{aligned} f(a, b, c, d, e) = & !a!bc!de + !a!bcd!e + !a!bcde + !ab!c!de + !ab!cd!e + !ab!cde + a!b!c!de \\ & + a!b!cd!e + a!b!cde + a!bc!de + a!bcd!e + a!bcde + ab!c!de + ab!cd!e \\ & + ab!cde + abc!de + abcd!e + abcde. \end{aligned}$$

A árvore de execução completa do algoritmo proposto é mostrada na Figura 4, onde arestas sólidas denotam chamadas recursivas e arestas pontilhadas representam a informação retornada por chamadas recursivas que encontraram uma decomposição.

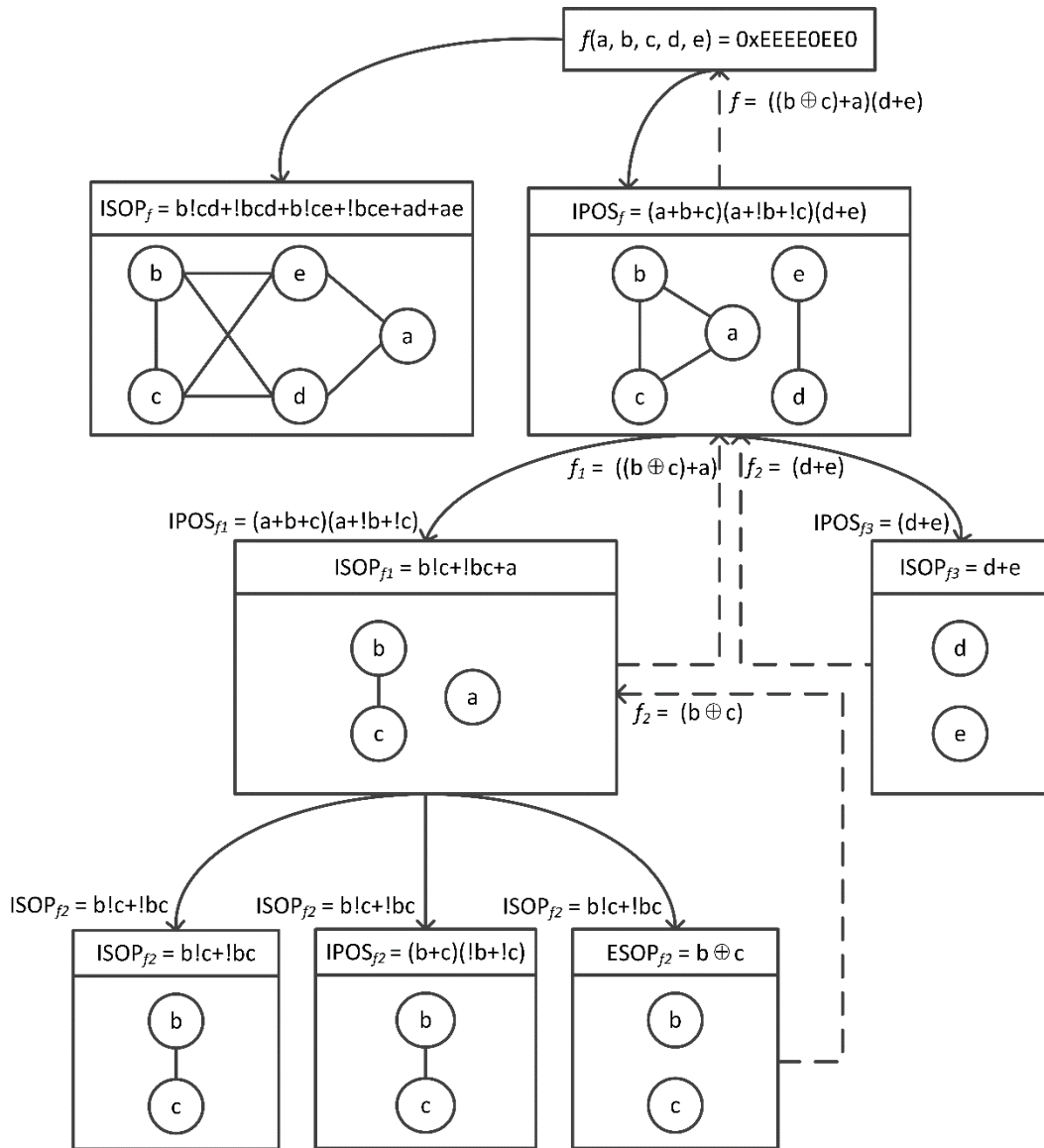


Figura 4 – Uma árvore de execução completa do algoritmo proposto.

### Resultados experimentais

Uma ferramenta para síntese de funções DSD foi apresentada. A ferramenta proposta usa ferramentas como *ESPRESSO* (BRAYTON ET AL, 1984) e *EXORCISM-4* (MISHCHENKO; PERKOWSKI, 2001) de forma a prover formas ISOP, IPOS e ESOP, respectivamente. A plataforma utilizada para a execução deste teste foi um Sistema Linux com Intel Core i5 2400 e com 4 GB de memória principal.

De forma a demonstrar a eficiência e precisão da abordagem proposta, dois experimentos foram executados. O primeiro usa todas as funções DSD de até 6 entradas. O segundo experimento foi feito selecionando PLAs de referência disponíveis em (*ESPRESSO BOOK*

EXAMPLES). Em todos estes experimentos, o tempo total levado em consideração inclui leitura e escrita dos arquivos bem como a troca de contexto entre chamadas de distintas ferramentas (ESPRESSO e EXORCISM-4).

Todas as funções DSD de até 6 entradas foram agrupadas em classe de equivalência através de permutação de entradas (classes-P). Resultados considerando a decomposição dessas funções são apresentados na Tabela 2. A coluna *Tempo* mostra o tempo necessário (em segundos) para a execução de todas as classes. O método proposto encontrou decomposições DSD para todas as funções testadas com sucesso.

Tabela 2 – Resultados para um conjunto de funções composto de funções DSD de até 6 variáveis, agrupado por equivalência através de permutação de entrada (classes-P).

Entradas	Classes P	Tempo (s)	Pior tempo (s)	Tempo Médio (s)
2	8	0.2	0.06	0.02
3	36	1.5	0.11	0.04
4	206	13	0.18	0.06
5	1,259	107	0.27	0.09
6	8,448	909	0.31	0.11

Um segundo experimento foi executado em exemplos obtidos em (ESPRESSO BOOK EXAMPLES). Os resultados são mostrados na Tabela 3, onde a segunda e a terceira coluna denotam o número de entradas e de saídas primárias, respectivamente. A coluna *Saídas decompostas* reporta o número de saídas às quais uma decomposição DSD foi encontrada com sucesso. A coluna *Tempo* reporta o tempo total para execução do método.

Tabela 3 – Circuitos selecionados do benchmark (ESPRESSO BOOK).

Circuito	Entradas primárias	Saídas primárias	Saídas decompostas	Tempo (s)
5xp1	7	10	4	0.97
9sym	9	1	0	0.07
alu4	14	8	0	0.71
apex1	45	45	12	3.37
apex2	39	3	0	3.09
apex3	54	50	18	3.31
apex4	9	19	1	1.21
apex5	117	88	9	7.13
b12	15	9	2	0.52
bw	5	28	7	1.51
clip	9	5	0	0.24
con1	7	2	0	0.08
cordic	23	2	0	5.05
cps	24	109	51	9.02
duke2	22	29	8	2.08
e64	65	65	65	0.88
ex1010	10	10	0	0.77
ex4	128	28	14	1.36
ex5	8	63	35	1.91
inc	7	9	2	0.57
misex1	8	7	0	0.35
misex2	25	18	12	0.7
misex3	14	14	0	1.02
misex3c	14	14	0	0.8
mytest	2	1	1	0.01
pdic	16	40	12	2.48
rd53	5	3	1	0.13
rd73	7	3	1	0.14
rd84	8	4	2	0.17
sao2	10	4	0	0.4
seq	41	35	2	5.03
spla	16	46	12	3.76
squar5	5	8	3	0.41
t481	16	1	1	0.48
table3	14	14	0	1.09
table5	17	15	0	1.62
xor5	5	1	1	0.04
Z5xp1	7	10	4	0.83
Z9sym	9	1	0	0.05
ex1010	10	10	0	1.06
ex4	128	28	14	1.47
ibm	48	17	0	0.82
jbp	36	57	31	3.71
mainpla	27	54	0	5.43
misg	56	23	22	0.35
mish	94	43	43	0.89
misj	35	14	14	0.37
pdic	16	40	12	2.5
shift	19	16	1	0.69
signet	39	8	4	7.6
soar	83	94	47	7.19
test2	11	35	0	5.17
test3	10	35	0	2.52
ti	47	72	21	5.34
ts10	22	16	0	0.72
x7dn	66	15	0	1.27
xparc	41	73	11	13.87

#### 4. MÉTODO DE DECOMPOSIÇÃO BOTTOM-UP PARA SÍNTESE DE FUNÇÕES DISJUNTAS DE SUPORTE

Esta seção apresenta uma nova abordagem para decomposição baseado na análise de cofatores e diferença Booleana (CALLEGARO ET AL, 2015b). ). Dois testes simples resultam em condições necessárias e suficientes para identificar decomposições AND e XOR. Estes testes foram apresentados por Kodandapandi, em (KODANDAPANDI; SETH, 1978), no contexto de diagramas de decomposição (*decomposition charts*). Estes testes são revisitados, e uma nova e eficiente abordagem baseada em cofatores é apresentada. Além de decomposições AND e XOR, uma decomposição para multiplexador (MUX) é apresentada. Testes necessários e suficientes para detecção de decomposição MUX para um número arbitrário de entradas também é apresentado. Por fim, um algoritmo que precisa no máximo  $O(n \cdot \log n)$  cofatores e  $O(n)$  testes de equivalência é apresentado. Resultados experimentais demonstram a eficiência do método proposto, quando comparado com estratégias de decomposição que são estado-da-arte.

##### Definições

Dada uma função  $F(X)$ , a operação de cofator consiste no assinalamento de um valor binário  $c_i \in B$ , a uma variável de entrada  $x_i \in X$ . A operação de cofator é denotada por  $F^{x_i=c_i}$ . O cofator negativo (positivo) de uma variável  $x_i$  é denotado por  $F^{x_i=0}$  ( $F^{x_i=1}$ ). A diferença Booleana de uma função  $F$  com relação a uma variável  $x_i$ , é denotado por  $F_{x_i}$  e é definido como  $F_{x_i} = F^{x_i=0} \oplus F^{x_i=1}$ .

##### Regras de decomposição

**Teorema (Decomposição AND).** Deixe  $F(X_1, X_2)$  ser uma função Booleana com  $X_1 = \{x_i, x_j\}$ ,  $X_1 \cap X_2 = \emptyset$ . Existe uma função  $H(z_1, X_2)$ , onde  $z_1 = G(X_1) = x_i^{c_i} \cdot x_j^{c_j}$ , de forma que  $F(X_1, X_2) = H(G(X_1), X_2)$ , se e somente se:

$$F^{x_i=\bar{c}_i} = F^{x_j=\bar{c}_j} \quad (83)$$

**Teorema (Decomposição XOR).** Deixe  $F(X_1, X_2)$  ser uma função Booleana com  $X_1 = \{x_i, x_j\}$ ,  $X_1 \cap X_2 = \emptyset$ . Existe uma função  $H(z_1, X_2)$ , onde  $z_1 = G(X_1) = x_i^{c_i} \oplus x_j^{c_j}$ , de forma que  $F(X_1, X_2) = H(G(X_1), X_2)$ , se e somente se:

$$F_{x_i} = F_{x_j} \quad (84)$$



**Teorema (Decomposição MUX).** Deixe  $F(X_1, X_2)$  ser uma função Booleana com  $X_1 = S \cup D$ ,  $S = \{x_s\}$ ,  $D = \{x_i, x_j\}$ ,  $S \cap D = \emptyset$ ,  $X_1 \cap X_2 = \emptyset$ . Existe uma função  $H(z_1, X_2)$ , onde  $z_1 = G(X_1) = MUX(S, D) = \overline{x_s} \cdot x_i + x_s \cdot x_j$ , de forma que  $F(X_1, X_2) = H(G(X_1), X_2)$ , se e somente se todas as equações (85)-(90) são satisfeitas:

$$F_{x_i} \neq F_{x_j} \quad (85)$$

$$F_{x_i x_j} = 0 \quad (86)$$

$$(F_{x_i})^{x_s=1} = 0 \quad (87)$$

$$(F_{x_j})^{x_s=0} = 0 \quad (88)$$

$$(F^{x_i=0, x_j=0})_{x_s} = 0 \quad (89)$$

$$(F^{x_i=1, x_j=1})_{x_s} = 0 \quad (90)$$

## Resultados experimentais

Esta seção apresenta resultados do algoritmo proposto, que foi implementado na ferramenta ABC (BERKELEY, 2016). A plataforma utilizada para a execução deste teste foi um Sistema Linux com Intel Core i5 2400 e com 4 GB de memória principal.

Uma comparação do método proposto com relação ao método estado-da-arte disponível na ferramenta ABC foi realizada. Os métodos comparados podem ser executados através dos comandos *testdec -A 3* (MISHCHENKO; STEINBACH; PERKOWSKI, 2001) e *testdec -A 4*, que é uma nova versão do método apresentado em (MISHCHENKO; STEINBACH; PERKOWSKI, 2001). Os métodos são comparados utilizando diversos conjuntos de funções como entrada. Em todos os testes, todos os métodos deram soluções corretas e idênticas.

O primeiro experimento foi realizado utilizando-se um conjunto de funções DSD (CALLEGARO, 2016). O primeiro conjunto de funções contém todas as funções *full-DSD* de até 6 variáveis, como exibido na primeira linha da

Tabela 4. A segunda linha representa um conjunto aleatório de funções *full-DSD* de até 7 entradas. O método proposto é mais eficiente do que o algoritmo apresentado em (MISHCHENKO; STEINBACH; PERKOWSKI, 2001), considerando ambas as versões – original e versão mais recente.

Tabela 4 – Comparação de métodos de decomposição DSD considerando dois conjuntos de funções *full-DSD*.

Entradas	Funções	<i>testdec -A 3</i>	<i>testdec -A 4</i>	Método proposto
6	2,311,640	5.49 s	6.76 s	1.5 s
7	2,744,691	10.7 s	10.15 s	2.64 s

O segundo experimento foi realizado levando em conta as mesmas funções de referência retiradas de (HUANG ET AL, 2013). Os resultados apresentados na Figura 5 comparam o tempo de execução (em segundos) após a aplicação de cada método sobre o conjunto de funções. O método proposto é mais rápido para funções de até 10 variáveis de entrada. Para funções com mais de 12 variáveis, o método descrito em (MISHCHENKO; STEINBACH; PERKOWSKI, 2001) e com melhorias apresenta um tempo de execução melhor.

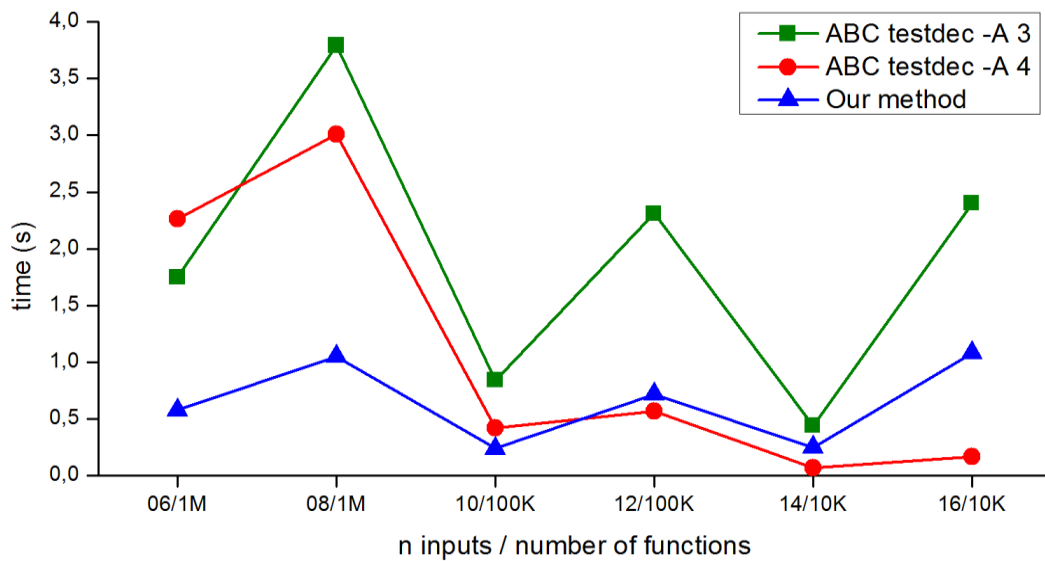


Figura 5 - Comparação entre o método proposto e os métodos estado-da-arte para decomposição DSD disponíveis na ferramenta ABC. O tempo de execução do método proposto é representado por triângulos azuis.

## 5. MÉTODO DE SÍNTESE PARA FUNÇÕES READ-POLARITY-ONCE

Esta seção apresenta uma comparação entre classes de funções *read-once* (RO), decomposição disjunta de suporte (DSD – *disjoint support decomposition*) e *read-polarity-once* (RPO). Uma primeira análise foi feita considerando todo o universo de funções Booleanas até oito entradas. Estes resultados são exibidos na Tabela 5. O número de funções em cada classe é apresentado em suas respectivas colunas. A quinta coluna mostra o número de funções que são tanto DSD quanto RPO. Cada linha na Tabela 5 representa o número total de funções por classe até  $n$  entradas, onde  $2 \leq n \leq 8$ . O número de funções possíveis com  $n$  entradas é  $2^{2^n}$ .

Tabela 5 – Enumeração de classes de funções Booleanas: *read-once* (RO), decomposição disjunta de suporte (DSD) e *read-polarity-once* (RPO).

Entradas	RO	DSD	RPO	DSD $\cap$ RPO	$\frac{DSD \cap RPO}{DSD}$
2	12	14	14	14	100%
3	94	150	228	148	99%
4	1,144	2,678	20,748	2,492	93%
5	19,994	68,966	6,814,286	57,894	84%
6	456,774	2,311,640	3,934,102,220	1,699,626	74%
7	12,851,768	95,193,064	-	-	-
8	429,005,426	4,645,069,336	-	-	-

Outra comparação entre classes foi realizada considerando circuitos de referência MCNC (IWLS, 2005). Resultados confirmam que o número de funções RPO é representativamente maior que funções RO e DSD. Isto significa que funções RPO são também importantes em circuitos industriais, e que um algoritmo de síntese exata para funções RPO pode melhorar a qualidade de circuitos digitais.

Além da comparação, esta seção apresenta um novo método para síntese de funções RPO (CALLEGARO ET AL, 2013). O conceito apresentado neste algoritmo é mais simples que o apresentado em (CALLEGARO ET AL, 2012), além ser consideravelmente mais rápido. O método proposto foi capaz de eficientemente encontrar soluções ótimas com até 16 literais, enquanto outros métodos não (YOSHIDA; FUJITA, 2011) (MARTINS ET AL, 2012).

## Resultados experimentais

O primeiro experimento foi feito usando o conjunto de todas as funções de 5 entradas agrupadas em classe de equivalência NPN (funções equivalentes através de negação / permutação de entradas e negação de saída). Este conjunto de funções contém 616,125 classes NPN de até 5 entradas. A plataforma utilizada para a execução deste teste foi um Sistema Linux com Intel Core i5 2400 e com 4 GB de memória principal.

Para rodar o algoritmo proposto para todo o conjunto de 616,125 funções o tempo de execução foi de 44 segundos. O pior caso de otimização levou 1ms e, na média, cada função levou 70  $\mu$ s para ser sintetizada. A implementação feita baseada no algoritmo X-Factor (MINTZ, GOLUMBIC, 2005) levou 50 minutos para rodar, resultando em um total de total 12,530,011 literais. O algoritmo FC (MARTINS ET AL, 2012) resultou em 11,292,029 literais e precisou de 6 horas para executar. É importante notar que ambos algoritmos (FC e X-Factor) foram projetados para tratar de funções arbitrárias, enquanto o algoritmo aqui proposto foi projetado para sintetizar apenas funções RPO.

Resultados da avaliação da eficiência do algoritmo proposto é mostrado na Tabela 6, considerando apenas o subconjunto de 1,462 funções RPO, extraídos do conjunto de classes-NPN de até 5 variáveis. O método proposto apresenta resultados melhores tanto em tempo de execução quando em contagem de literais, quando comparado com os métodos *Quick Factor* (QF) (SETOVICH ET AL, 1992), *Good Factor* (GF) (SETOVICH ET AL, 1992), *Functional Composition* (FC) (MARTINS ET AL, 2012) X-Factor (MINTZ, GOLUMBIC, 2005).

Tabela 6 – Número total de literais e tempo de execução para obter formas fatoradas para 1,462 funções RPO utilizando diferentes abordagens.

	<b>QF</b>	<b>GF</b>	<b>FC</b>	<b>X-Factor</b>	<b>RPO 2013</b>	<b>Este trabalho</b>
<b>Literais</b>	16,086	15,671	13,754	13,253	13,064	13,064
<b>Tempo</b>	1.9s	2.3s	21s	7.1s	5.7s	0.7s

No segundo experimento, um estudo utilizando circuitos MCNC foi realizado (IWLS 2005). Foram extraídas funções de até 10 entradas dos circuitos através de enumeração de cortes-k (CHATTERJEE; MISHCHENKO; BRAYTON, 2006), (MACHADO ET AL, 2012). Infelizmente os métodos FC e X-Factor foram muito lentos e não foi possível se obter

soluções para os circuitos em um tempo factível. O algoritmo proposto levou 36s, 2m36s e 6m53s para fatorar funções extraídas dos circuitos de 6, 8 e 10 entradas, respectivamente. Como é possível ver na Tabela 7, funções RPO também são bastante frequentes em circuitos de referência. Isto mostra que funções RPO são de especial interesse em aplicações como circuitos digitais.

O último experimento consiste na análise de funções não-RPO considerando os seguintes conjuntos de funções: todas as funções de até três e quatro variáveis, todas as funções de até cinco entradas agrupadas por equivalência através de negação / permutação de entradas (NPN-5) e negação de saída, todas as funções DSD de até seis variáveis, funções DSD, DSD-parciais e não-DSD extraídas de (HUANG ET AL, 2013). Este estudo é para avaliar quão frequentes funções não-RPO podem ser decompostas usando uma expansão RPO de distância-1, chamadas 1-dist RPO. Os resultados são mostrados na Tabela 8. O número de funções que têm soluções 1-dist RPO através de expansões de Shannon, Davio ou Quantifier-Based é exibido nas colunas Shannon, Davio e Quantifier-Based, respectivamente. A coluna 1-dist RPO mostra o número e funções que tem ao menos uma das decomposições acima mencionadas. Finalmente, a coluna RPO + 1-dist RPO representa o número de funções que são RPO ou tem uma decomposição 1-dist RPO.

Como esperado, todas as funções de até 3 entradas são RPO ou podem ser representadas por uma expansão 1-dist RPO. Curiosamente, 99% das funções de até 4 variáveis são RPO ou tem uma decomposição 1-dist RPO. Para o conjunto de funções NPN de 5 entradas e para funções DSD de até 6 variáveis, este número é 64% e 98%, respectivamente. Para os conjuntos de funções DSD, parcial-DSD e não-DSD de 6 até 16 entradas a o número de funções que tem decomposição 1-dist RPO também é bastante frequente.

Tabela 7 – Decomposição de circuitos em funções de até K entradas, com K variando de 6 a 10. O número de funções *read-once* (RO), decomposição disjunta de suporte (DSD) e *read-polarity-once* (RPO) é apresentado.

Circuitos	K = 6					K = 8					K = 10				
	Total	RO	DSD	RPO	Tempo (ms)	Total	RO	DSD	RPO	Tempo (ms)	Total	RO	DSD	RPO	Tempo (ms)
9symml	41	12	16	31	389	9	1	2	2	182	1	0	0	0	46
alu2	109	65	75	90	165	39	10	17	19	749	6	1	3	3	5718
alu4	194	103	125	148	131	115	40	60	64	12632	48	6	12	15	5349
apex6	157	44	64	153	51	144	30	50	139	54	117	22	41	110	75
apex7	55	35	47	52	16	45	24	36	42	16	42	22	34	39	21
b1	4	2	3	4	0	4	2	3	4	0	4	2	3	4	1
c8	28	5	11	28	6	22	5	5	22	6	20	3	3	20	8
cc	21	13	13	21	5	20	12	12	20	5	20	12	12	20	5
cht	36	0	0	36	7	36	0	0	36	7	36	0	0	36	9
cm150a	6	1	1	1	1	5	0	0	0	1	5	0	0	0	2
cm151a	4	0	0	2	1	4	0	0	2	0	3	0	0	0	1
cm152a	3	0	0	1	0	2	0	0	0	0	2	0	0	0	0
cm162a	10	6	10	10	3	7	3	6	6	12	6	2	2	6	7
cm163a	7	3	3	7	2	6	2	2	6	3	5	1	1	5	3
cm42a	10	10	10	10	2	10	10	10	10	2	10	10	10	10	2
cm82a	3	0	1	0	1	3	0	1	0	0	3	0	1	0	1
cm85a	6	0	2	2	2	6	0	2	2	2	3	0	1	1	2
cmb	11	8	8	11	3	8	5	5	8	3	7	5	5	7	191
comp	22	8	15	15	6	11	3	7	7	6	11	3	7	7	6
cordic	9	5	7	7	3	8	5	6	6	4	5	0	1	1	7
count	24	8	8	24	6	21	4	5	21	8	20	3	5	20	12
cu	16	14	14	16	2	13	8	8	13	3	12	8	8	11	3
dalu	235	36	75	134	81	186	20	45	63	386	151	16	16	44	18569
decod	16	16	16	16	3	16	16	16	16	3	16	16	16	16	3
des	1035	309	534	931	1892	778	58	305	563	2242	350	6	29	206	239
example2	94	62	67	89	11	79	25	49	48	352	72	30	35	60	3037
f51m	21	7	9	11	3	8	1	3	3	2	8	1	3	3	2
frg1	26	21	21	21	4	24	16	16	17	6	17	8	8	9	6
i10	582	345	432	544	548	486	265	340	432	1719	464	230	273	371	29981
i1	18	16	17	18	2	17	15	16	17	2	17	15	16	17	1
i2	66	65	65	66	7	34	32	32	34	8	28	26	26	26	10
i3	42	42	42	42	6	22	22	22	22	5	22	22	22	22	6
i4	62	62	62	62	10	38	38	38	38	10	34	34	34	34	12
i5	76	76	76	76	11	66	66	66	66	12	68	68	68	68	18
i6	67	0	0	1	11	67	0	0	1	11	67	0	0	1	13
i7	67	1	3	3	9	67	1	3	3	10	67	1	3	3	10
i8	284	123	123	284	42	170	42	42	168	40	242	78	78	187	109
i9	229	171	171	227	39	75	5	5	10	29	68	0	0	1	44
k2	543	525	526	536	77	486	434	439	474	31055	374	243	246	315	264658
lal	25	15	25	25	5	23	14	23	23	5	21	12	21	21	7
majority	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
my_adder	25	0	8	1	7	23	0	6	1	8	21	0	4	1	12
pair	338	199	242	316	150	252	115	152	208	3153	239	110	147	187	2510
parity	5	0	5	0	1	3	0	3	1	1	2	0	2	0	1
pcler8	24	17	23	24	3	22	15	18	22	4	21	14	18	21	4
pcle	13	6	13	13	3	12	4	5	12	3	11	4	11	11	3
pm1	16	14	15	16	2	14	12	13	14	2	13	11	12	13	2
sct	21	13	18	21	4	18	8	15	18	3	17	7	14	17	5
tcon	16	8	8	16	1	16	8	8	16	0	16	8	8	16	0
term1	42	23	29	34	10	34	18	23	26	56	20	6	11	12	24370
ttt2	35	15	22	31	23	24	4	11	17	4277	23	3	10	16	138
unreg	16	0	0	16	2	16	0	0	16	3	16	0	0	16	2
vda	283	249	252	272	32612	197	120	120	136	98615	131	47	48	72	57509
x1	89	79	79	85	10	74	63	63	68	12	61	45	45	54	20
x2	14	12	13	13	1	10	7	8	8	1	7	4	5	5	1
x3	167	57	78	167	21	138	29	50	135	24	111	17	37	107	30
x4	109	89	96	108	13	83	41	48	82	16	78	55	62	77	23
z4ml	6	0	2	1	0	4	0	1	0	1	4	0	1	0	1
<b>Total</b>	5484	3015	3600	4889	36426	4121	1678	2241	3207	155771	3264	1237	1478	2344	412825
<b>Razão</b>	1	0.55	0.66	0.89	-	1	0.41	0.54	0.78	-	1	0.38	0.45	0.72	-

Tabela 8 – Análise de funções RPO e não-RPO. As colunas Shannon, Davio e Quantifier-Based mostram o número de funções dos respectivos benchmarks que tem decomposição 1-dist RPO usando as expansões de Shannon, Davio e Quantifier-Based, respectivamente. A coluna 1-dist RPO mostra o número de funções que tem ao menos uma das decomposições anteriores. Finalmente, a coluna RPO + 1-dist RPO representa o número de funções que são RPO ou tem uma decomposição 1-dist RPO.

Benchmark	Funções	RPO (%)	Shannon (%)	Davio (%)	Quantifier-Based (%)	1-dist RPO (%)	RPO + 1-dist RPO (%)
All 3-in functions	256	230 (89.8%)	26 (10.2%)	24 (9.4%)	26 (10.2%)	26 (10.2%)	256 (100.0%)
All 4-in functions	65,536	20,750 (31.7%)	42,496 (64.8%)	43,720 (66.7%)	42,688 (65.1%)	44224 (67.5%)	64,974 (99.1%)
NPN 5-in	616,125	1,421 (0.2%)	177,507 (28.8%)	282,288 (45.8%)	317,378 (51.5%)	398307 (64.6%)	399,727 (64.9%)
All DSD up to 6-in	2,311,640	1,699,626 (73.5%)	544,680 (23.6%)	574,640 (24.9%)	585,288 (25.3%)	585480 (25.3%)	2,285,108 (98.9%)
Full-DSD 6-in	1,000,000	992,754 (99.3%)	4,614 (0.5%)	4,788 (0.5%)	4,788 (0.5%)	4788 (0.5%)	997,542 (99.8%)
Full-DSD 8-in	1,000,000	925,190 (92.5%)	5,267 (0.5%)	5,735 (0.6%)	5,716 (0.6%)	5740 (0.6%)	930,930 (93.1%)
Full-DSD 10-in	100,000	98,117 (98.1%)	968 (1.0%)	1,054 (1.1%)	1,043 (1.0%)	1054 (1.1%)	99,171 (99.2%)
Full-DSD 12-in	100,000	97,496 (97.5%)	1,605 (1.6%)	1,612 (1.6%)	1,614 (1.6%)	1614 (1.6%)	99,110 (99.1%)
Full-DSD 14-in	10,000	9,104 (91.0%)	114 (1.1%)	126 (1.3%)	125 (1.3%)	126 (1.3%)	9,230 (92.3%)
Full-DSD 16-in	10,000	9,355 (93.6%)	65 (0.7%)	118 (1.2%)	118 (1.2%)	118 (1.2%)	9,473 (94.7%)
Partial-DSD 6-in	1,000,000	823,030 (82.3%)	165,327 (16.5%)	167,389 (16.7%)	168,430 (16.8%)	168950 (16.9%)	991,980 (99.2%)
Partial-DSD 8-in	1,000,000	683,462 (68.3%)	262,012 (26.2%)	283,453 (28.3%)	291,517 (29.2%)	294676 (29.5%)	978,138 (97.8%)
Partial-DSD 10-in	100,000	65,562 (65.6%)	32,017 (32.0%)	30,682 (30.7%)	32,500 (32.5%)	32949 (32.9%)	98,511 (98.5%)
Partial-DSD 12-in	100,000	50,818 (50.8%)	40,838 (40.8%)	39,327 (39.3%)	42,329 (42.3%)	44061 (44.1%)	94,879 (94.9%)
Partial-DSD 14-in	10,000	6,861 (68.6%)	2,001 (20.0%)	2,189 (21.9%)	2,251 (22.5%)	2297 (23.0%)	9,158 (91.6%)
Partial-DSD 16-in	10,000	4,653 (46.5%)	2,900 (29.0%)	3,148 (31.5%)	3,451 (34.5%)	3519 (35.2%)	8,172 (81.7%)
Non-DSD 6-in	1,000,000	134,275 (13.4%)	809,998 (81.0%)	815,630 (81.6%)	827,986 (82.8%)	834525 (83.5%)	968,800 (96.9%)
Non-DSD 8-in	1,000,000	27,280 (2.7%)	534,261 (53.4%)	536,794 (53.7%)	590,349 (59.0%)	663394 (66.3%)	690,674 (69.1%)
Non-DSD 10-in	100,000	83 (0.1%)	26,722 (26.7%)	23,068 (23.1%)	26,671 (26.7%)	32540 (32.5%)	32,623 (32.6%)
Non-DSD 12-in	100,000	364 (0.4%)	12,456 (12.5%)	13,432 (13.4%)	14,924 (14.9%)	17220 (17.2%)	17,584 (17.6%)
Non-DSD 14-in	10,000	22 (0.2%)	940 (9.4%)	900 (9.0%)	1,073 (10.7%)	1263 (12.6%)	1,285 (12.9%)
Non-DSD 16-in	10,000	20 (0.2%)	350 (3.5%)	390 (3.9%)	380 (3.8%)	460 (4.6%)	480 (4.8%)
Total	9,653,557	5,650,474 (58.5%)	2,667,164 (27.6%)	2,842,366 (29.4%)	2,960,645 (30.7%)	3,137,331 (32.5%)	8,787,805 (91.0%)

## 6. CONCLUSÕES

O problema de fatoração e decomposição para funções genéricas é um problema  $\Sigma_2^P$ -complete. Algoritmos eficientes e exatos podem ser criados para classes de funções Booleanas como *read-once* (RO), decomposição disjunta de suporte (DSD) e *read-polarity-once* (RPO).

Esta tese discutiu métodos de síntese lógica que são focados em classes específicas de funções Booleanas. Quatro métodos de síntese de funções Booleana foram apresentados. A primeira contribuição foi um método para síntese de funções *read-once*. O método é baseado em uma estratégia de divisão-e-conquista. Dado um BDD com  $m$  nodos e  $n$  entradas, o método **RO\_BY\_COFACTOR** tem uma complexidade de pior caso  $O(mn)$ . Este método foi implementado em na ferramenta de síntese lógica chamada SwitchCraft (CALLEGARO ET AL, 2010).

A segunda e terceira contribuições são métodos para síntese de funções DSD. (CALLEGARO ET AL, 2015a). Uma abordagem *top-down* checa se existe uma decomposição OR, AND ou XOR baseado em entradas de soma-de-produtos, produto-de-somas e soma-exclusiva-de-produtos, respectivamente. Este método também está disponível na ferramenta SwitchCraft. O outro método é uma abordagem *bottom-up* e é baseado na análise de diferenças Booleanas e cofactores (CALLEGARO ET AL, 2015b). O método proposto precisa de  $O(n \cdot \log n)$  cofactores e  $O(n)$  testes de equivalências. O algoritmo foi implementado tanto na ferramenta ABC como na ferramenta SwitchCraft.

A última contribuição é um novo método para síntese de funções RPO (CALLEGARO ET AL, 2013). O método é baseado no conceito de conjunto de transições positivas e negativas das variáveis de entrada. O método é capaz de detectar se dois literais devem ser agrupados utilizando uma operação lógica AND ou OR pela análise dos conjuntos de transições. O método também foi implementado nas ferramentas ABC (comando *test\_rpo*) e SwitchCraft.