Read-polarity-once functions

Thesis presented in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Prof. Dr. André Inácio Reis
Advisor

Prof. Dr. Renato Perez Ribas
Co-advisor

Porto Alegre, July 2012
Callegaro, Vinicius


65 f.:il.


ACKNOWLEDGEMENTS

I would like to dedicate this thesis to my parents Aldo and Marta, to my sister Thalita and to my fiancée Monica, who were essential to me during the master years.

I am grateful to my advisor Andre Inácio Reis and my co-advisor Renato Perez Ribas for their knowledge and effort to make this thesis possible.

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 Raul Fernando Weber, Sérgio Bampi and Marcelo de Oliveira Johann who helped me to improve this work in several ways. Special acknowledgments go to Raul Fernando Weber by his pioneering classes in the study of logic synthesis in our university.

The research presented in the thesis was supported by Nangate under a Nangate/UFRGS research agreement, by CAPES and CNPq Brazilian funding agencies, by FAPERGS under grant 11/2053-9 (Pronem), and by the European Community's Seventh Framework Programme under grant 248538 – Synaptic.
# CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS .................................................. 6  
LIST OF FIGURES ............................................................................................. 7  
LIST OF TABLES .............................................................................................. 9  
ABSTRACT ...................................................................................................... 10  
RESUMO .......................................................................................................... 11  
1 INTRODUCTION .......................................................................................... 12  
1.1 Motivation .................................................................................................. 12  
1.2 Objective.................................................................................................... 14  
1.3 Thesis Organization .................................................................................. 14  
2 BOOLEAN LOGIC CONCEPTS ...................................................................... 15  
2.1 Sets, membership and basic operations ................................................. 15  
2.2 Boolean functions .................................................................................... 16  
2.2.1 Representing Boolean functions ............................................................. 16  
2.2.2 Operations on completely specified Boolean functions ....................... 18  
2.2.3 Operations on incompletely specified Boolean functions ..................... 19  
2.2.4 Cofactor over Boolean functions ........................................................... 20  
2.2.5 Unateness .............................................................................................. 21  
2.3 Logic equations ....................................................................................... 22  
3 READ-ONCE BOOLEAN FUNCTIONS .................................................... 24  
3.1 Definition .................................................................................................. 24  
3.2 Previous work .......................................................................................... 24  
3.3 IROF method .......................................................................................... 25  
3.4 JPHI method .......................................................................................... 27  
3.5 IROF and JPHI comparison ........................................................................ 30  
4 FACTORING INCOMPLETELY SPECIFIED BOOLEAN FUNCTIONS INTO READ-ONCE EQUATIONS ...................................................... 33  
4.1 ISF2RO method ....................................................................................... 36  
5 READ-POLARITY-ONCE FUNCTIONS ................................................... 40
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDD</td>
<td>Binary Decision Diagram</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>CSF</td>
<td>Completely Specified Boolean Function</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>GF</td>
<td>Good Factor</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>IPOS</td>
<td>Irredundant Sum-Of-Products</td>
</tr>
<tr>
<td>IROF</td>
<td>Is Read-Once Function</td>
</tr>
<tr>
<td>ISF</td>
<td>Incompletely Specified Boolean Functions</td>
</tr>
<tr>
<td>ISF2RO</td>
<td>Incompletely Specified Boolean Functions to \textit{read-once}</td>
</tr>
<tr>
<td>ISF2RPO</td>
<td>Incompletely Specified Boolean Functions to \textit{read-polarity-once}</td>
</tr>
<tr>
<td>ISOP</td>
<td>Irredundant Sum-Of-Products</td>
</tr>
<tr>
<td>JPHI</td>
<td>John P. Hayes Improved</td>
</tr>
<tr>
<td>LA</td>
<td>Logic Arrangement</td>
</tr>
<tr>
<td>NPN</td>
<td>Negation-Permutation-Negation</td>
</tr>
<tr>
<td>POS</td>
<td>Product-Of-Sums</td>
</tr>
<tr>
<td>QF</td>
<td>Quick Factor</td>
</tr>
<tr>
<td>RO</td>
<td>Read-Once</td>
</tr>
<tr>
<td>ROBDD</td>
<td>Reduced and Ordered Binary Decision Diagram</td>
</tr>
<tr>
<td>RPO</td>
<td>Read-Polarity-Once</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SOP</td>
<td>Sum-Of-Products</td>
</tr>
<tr>
<td>SPFD</td>
<td>Sets of Pairs of Functions to be Distinguished</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very-High-Speed Integrated Circuits</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1: A Karnaugh map representation of \( f(x_1, x_2, x_3) = 11001001_2 \). .......... 17
Figure 2.2: Binary decision diagram (BDD) representation of a Boolean function. .......... 17
Figure 2.3: A reduced and ordered binary decision diagram (ROBDD) of a Boolean function. ................................................................. 17
Figure 2.4: Karnaugh map representation of \( f = 11101X1011101X102 \) (a) and \( g = 1110111101X010X02 \) (b). Dashes mean don’t-care (X) terms. ......................... 20
Figure 2.5: The same Boolean function represented by an equation in sum-of-products (a) and in product-of-sums form (b). ......................................................... 23
Figure 2.6: Arbitrary Boolean function represented by a factored form. ......................... 23
Figure 3.1: The resulting graph for the ISOP \( f = x_1x_2 + x_1x_3x_4 + x_1x_3x_5 \). ........ 25
Figure 3.2: Pseudo algorithm for the coGraph_Rec step. .............................................. 26
Figure 3.3: Complement graph of Figure 3.1. In this step, the coGraph_Rec build the partial equation: \( f = x_1f_1(x_2, x_3, x_4, x_5) \). ................................................................. 26
Figure 3.4: Complement graph of Figure 3.3. The node \( x_1 \) was removed. In this step, the coGraph_Rec build the partial equation: \( f_1 = x_2f_2(x_3, x_4, x_5) \). ................................................................. 26
Figure 3.5: Complement graph of Figure 3.4. The node \( x_2 \) was removed. In this step, the coGraph_Rec build the partial equation: \( f_2 = x_3f_3(x_4, x_5) \). ................................................................. 27
Figure 3.6: Complement graph of Figure 3.5. The node \( x_3 \) was removed. In this step, the coGraph_Rec build the equation: \( f_3 = x_4 + x_5 \). ................................................................. 27
Figure 3.7: Equations of negative cofactors of \( f = x_1x_2x_3x_4 + x_1x_2x_3x_5 + x_4x_6 + x_5x_6 \). ........................................................................................................ 28
Figure 3.8: Partial decomposition after tying variables into \( \phi = x_1x_2x_3 \). ................. 28
Figure 3.9: Disappearance matrix of \( f = x_1x_2x_3x_4 + x_1x_2x_3x_5 + x_4x_6 + x_5x_6 \) Source: LEE; WANG, 2007. ................................................................. 29
Figure 3.10: Iterations of recognizing \( f = (x_1x_2x_3 + x_4)(x_4 + x_5) \). Source: LEE; WANG, 2007. ................................................................. 29
Figure 3.11: Iterations of recognizing \( f = (x_1x_2x_3 + x_4)(x_4 + x_5) \). Source: LEE; WANG, 2007. ................................................................. 30
Figure 3.12: The average (in ms) runtime to synthesize the benchmark of 3503 read-once functions, grouped by number of inputs. ......................................................... 32
Figure 4.1: Example of an ISF that can be factorized into a read-once form. ................. 34
Figure 4.2: Karnaugh map for the SOP \( f = x_3x_4x_5x_6 + x_0x_1x_3 + x_0x_1x_2 \). This function was synthesized by the Quine-McCluskey’s method and cannot be represented by a read-once formula ................................................................. 35
Figure 4.3 Karnaugh map for the SOP \( f = x_0x_1x_2 + x_0x_1x_3 + x_0x_2x_4x_6 + x_0x_3x_4x_6 + x_1x_2x_5 + x_1x_3x_5 + x_2x_4x_5x_6 + x_3x_4x_5x_6 \). This function was synthesized by the ISF2RO method (section 4.1) and can be represented by a read-once formula \( f = (x_0 + x_5)(x_3 + x_2)(x_1 + x_6x_4) \). ................................................................. 35
Figure 4.4: *LogicArrangement* class diagram. ............................................................... 36
Figure 4.5: ISF2RO pseudo-algorithm. .......................................................................... 38
Figure 4.6: Worst case runtime (in ms) to synthesize functions from ISCAS ’85 benchmark grouped by input count. ............................................................................... 39
Figure 5.1: Comparison between the universe of *read-once* and *read-polarity-once* functions. ..................................................................................................................... 41
Figure 5.2: Pseudo-algorithm for the unatization process. ............................................. 43
Figure 5.3: First step of unatization process: split binate variables. ............................. 44
Figure 5.4: Second step of the unatization process: turn the variables into a positive unate behavior............................................................................................................. 44
Figure 5.5: Cofactors that are set to force the function to become positive unate. ...... 45
Figure 5.6: ISF2RPO flow chart.................................................................................... 46
Figure 5.7: The ISF $g$ obtained after the unatization process of the function $f=a \land b + \neg a \land b$. ................................................................................................................................. 46
Figure 5.8: Transformation of the remaining *don’t-care* terms and how it produces different read-once equations. ................................................................. 47
LIST OF TABLES

Table 1.1: The abundance of RO functions in a sample of Boolean function gathered from the ISCAS benchmarks ................................................................. 13
Table 2.1: Possible operations over sets ............................................................... 15
Table 2.2: A truth table representation of \( f(x_1,x_2,x_3)=11001001_2 \) .............. 16
Table 2.3: Truth table representation of \( \text{NOT} \) operator ................................. 18
Table 2.4 Truth table representation of \( \text{OR} \) operator ................................. 18
Table 2.5 Truth table representation of \( \text{AND} \) operator .................................. 18
Table 2.6 Truth table representation of \( \text{XOR} \) operator ................................. 19
Table 2.7: Summary of all binary operations over ISF in truth table data structure. 20
Table 3.1: Comparison regarding runtime (in seconds) required to synthesize an arbitrary benchmark of 9 read-once functions ............................................. 30
Table 3.2: Comparison of runtime between IROF and JPHI to synthesize the benchmark of 3503 read-once functions, grouped by number of inputs .................. 31
Table 3.3: behavior comparison between IROF and JPHI algorithm ..................... 32
Table 4.1: Don't-care assignments and results ..................................................... 36
Table 4.2: Expected initial main list ..................................................................... 37
Table 4.3: Grouped logic arrangements after first iteration .................................... 37
Table 4.4: Grouped logic arrangements after the second iteration ......................... 38
Table 5.1: Two states that must be fixed in the unatization process ....................... 42
Table 5.2: Total number of literals obtained and runtime when factoring process of 1,462 RPO functions using different approaches ......................................... 48
Table 5.4: Functions selected from the ISCAS’85 benchmarks using a greedy covering algorithm ........................................................... 49
Table 5.5: Runtime for synthesizing all K-Cuts functions (K=8) ............................. 50
Table 5.6: Total runtime for synthesizing functions selected by a greedy covering algorithm ........................................................... 50
Table 5.7: Set of 12 distinct functions given by Motiani where 10 were RPO functions. .................................................................................................................. 51
Tabela A.2.3: Funções selecionadas de circuitos ISCAS’85 utilizando um algoritmo de cobertura gulosa ............................................................. 63
Tabela A.2.4: Tempo de execução para a síntese de todos os K-Cuts (K=8) ............. 64
Tabela A.2.5: Tempo de execução para a síntese de funções selecionadas através da algoritmos de cobertura gulosa ......................................................... 64
ABSTRACT

Efficient exact factoring algorithms are limited to read-once functions, in which each variable appears once in the final Boolean equation. However, those algorithms present two main constraints: (1) they do not consider incompletely specified Boolean functions; and (2) they are not suitable for binate functions. To overcome the first drawback, it is proposed an algorithm that finds read-once formulas for incompletely specified Boolean functions, whenever possible. With respect to the second limitation, a domain transformation that splits existing binate variables into two independent unate variables is presented. Such domain transformation leads to incompletely specified Boolean functions, which can be efficiently factored by applying the proposed algorithm. The combination of both contributions gives optimal results for a novel broader class of Boolean functions named as read-polarity-once functions, where each polarity (positive or negative) of a variable appears at most once in the factored form. Experimental results over ISCAS'85 benchmark circuits have shown that read-polarity-once functions are significantly more frequent than read-once functions, for which many works have already been devoted in the literature.

Keywords: Boolean function, factoring, logic synthesis, incompletely specified functions, read-once function, read-polarity-once function.
Funções read-polarity-once

RESUMO

Algoritmos exatos para fatoração estão limitados a funções Booleanas read-once, onde cada variável aparece uma vez na equação final. No entanto, estes algoritmos apresentam duas restrições principais: (1) eles não consideram funções Booleanas incompletamente especificadas, e (2) eles não são adequados para as funções binate. Para superar o primeiro inconveniente, é proposto um algoritmo que encontra equações read-once para funções Booleanas incompletamente especificadas, sempre que possível, é proposto. Com respeito à segunda limitação, é apresentada uma transformação de domínio que divide variáveis binate existentes em duas variáveis unate independentes. Tal transformação de domínio conduz a funções Booleanas incompletamente especificadas, que podem ser eficientemente fatoradas mediante a aplicação do algoritmo proposto. A combinação das duas contribuições dá resultados ótimos para uma nova classe de funções Booleanas chamada read-polarity-once, onde cada polaridade (positiva ou negativa) de uma variável aparece no máximo uma vez na forma fatorada da expressão Booleana. Resultados experimentais sobre circuitos ISCAS'85 mostrou que funções read-polarity-once são significativamente mais frequentes em circuitos reais quando comparado com a classe de funções read-once, a qual muitos trabalhos já foram dedicados na literatura.

1 INTRODUCTION

The circuit synthesis design flow is usually divided into three major steps: architectural synthesis, logic synthesis and physical synthesis (MICHELI, 1994). The architectural synthesis, often called high-level synthesis, consists of transforming an algorithmic description of a 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 HDL (Hardware Description Language) (e.g. VHDL and Verilog) format.

The logic synthesis process has been one of the most commercially successful areas of electronic design automation (EDA). This commercial success indicates that all the digital devices that we use in our day-to-day life have been designed with a set of logic synthesis tools. The logic synthesis task consists of several steps. These steps may be different according to the nature of the circuit, e.g. sequential or combinational. The goal of logic synthesis is to determine the primitive structure of a circuit, i.e., its gate level representation. It is typically divided in three phases: technology independent optimizations, technology mapping and technology dependent optimizations. The first one applies transformations that do not depend on the technology, but on the functional behavior of a Boolean network, e.g. factorization algorithms. Then, the technology mapping phase match 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.

The physical synthesis, or geometrical level synthesis, consists mainly of two major tasks: placement of blocks and routing of wires. The former distributes physically the cells whereas the latter performs the signal interconnections.

This work addresses the synthesis of Boolean functions in the scope of a digital circuit design flow, more precisely in the logic synthesis phase. However, the focus can be considered broader since this work proposes a new class of Boolean functions that may have application in different areas other than circuit synthesis.

1.1 Motivation

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, usually provided initially in sum-of-products (SOP) or product-of-sums (POS) form. For instance, \( f = a \cdot b + a \cdot c \cdot d + a \cdot c \cdot e \) can be factored into the logically equivalent equation \( f = a \cdot (b + c \cdot (d + e)) \).
Any given logic function can be represented by different factored equations. 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 electrical implementation of a Boolean function corresponds directly to its factored equation in terms of literals and device count. Generating an optimum factored form, i.e. the shortest length equation, is an NP-hard 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 XFactor (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 4 variables. 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).

Efficient exact algorithms exist for a sub-class of functions known as read-once (RO) functions (LEE; WANG, 2007; GOLUMBIC; MINTZ; ROTICS, 2008). A Boolean function is considered read-once if it can be represented in a factored form where each variable appears only once (GOLUMBIC; MINTZ; ROTICS, 2001). Reviewing the example above, the function $f = a \cdot (b + c \cdot (d + e))$ is RO. In Table 1.1 it is possible to see a sample of Boolean function gathered from the ISCAS benchmarks. The class of RO functions is of special interest in logic synthesis because they are quite frequent in circuit applications.

Table 1.1: The abundance of RO functions in a sample of Boolean function gathered from the ISCAS benchmarks.

<table>
<thead>
<tr>
<th>Circuit name</th>
<th>Logic level 2</th>
<th>Logic level 3</th>
<th>Logic level 4</th>
<th>Logic level 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. Eqns</td>
<td>RO eqns No. %</td>
<td>No. Eqns</td>
<td>RO eqns No. %</td>
</tr>
<tr>
<td>C432</td>
<td>142</td>
<td>79 55</td>
<td>142</td>
<td>91 64</td>
</tr>
<tr>
<td>C499</td>
<td>162</td>
<td>0 0</td>
<td>146</td>
<td>0 0</td>
</tr>
<tr>
<td>C880</td>
<td>430</td>
<td>312 73</td>
<td>278</td>
<td>157 57</td>
</tr>
<tr>
<td>C1355</td>
<td>496</td>
<td>326 66</td>
<td>409</td>
<td>168 41</td>
</tr>
<tr>
<td>C1908</td>
<td>822</td>
<td>810 98</td>
<td>734</td>
<td>569 77</td>
</tr>
<tr>
<td>C2670</td>
<td>1087</td>
<td>979 90</td>
<td>994</td>
<td>644 64</td>
</tr>
<tr>
<td>C3540</td>
<td>1588</td>
<td>1404 88</td>
<td>1550</td>
<td>1119 72</td>
</tr>
<tr>
<td>C5315</td>
<td>2108</td>
<td>1789 84</td>
<td>1844</td>
<td>1174 64</td>
</tr>
<tr>
<td>C6288</td>
<td>2159</td>
<td>1696 79</td>
<td>2147</td>
<td>603 28</td>
</tr>
<tr>
<td>C7552</td>
<td>3404</td>
<td>3290 97</td>
<td>3290</td>
<td>2410 73</td>
</tr>
<tr>
<td>Total</td>
<td>12398</td>
<td>10685 86</td>
<td>11534</td>
<td>6935 60</td>
</tr>
</tbody>
</table>


However, exact algorithms for RO functions present two important limitations: (1) they do not consider incompletely specified Boolean functions; and (2) they are not suitable for functions with binate variables (see Subsection 2.2.5). In this context, a
question can be raised: “Since RO functions are quite frequent in real circuit applications, suppose we extend the boundaries of this class, creating a new class of functions, in order to overcome the limitation (1) and (2). How broader is this new class with regard to the RO class? How much more frequent are the functions of this new class in real circuit applications?” These questions summarize our motivation.

1.2 Objective

The objective of this thesis is to transcend the boundaries of the RO class of functions. One way of doing that is to create a new broad class of Boolean functions, which can deal with binate functions and where factorized forms can be guaranteed minimal. An approach is to split the positive and negative phases of binate variables into two independent variables. This way, it is possible to represent the positive phase in one variable and the negative phase into another variable, being these two variables unate and independent each other. This domain transformation leads to ISBF. Therefore, it is interesting to develop a method that is able to factorize ISBF into RO forms, which are known by resulting minimal forms. The combination of both strategies can lead to a novel broader class of functions called read-polarity-once (RPO) functions, where each polarity (positive or negative) of a variable appears at maximum once in the minimal factored form, e.g. \( f = (!a*d+c)*(a+b) \). In this sense, this work aims to provide an efficient algorithm that guarantees minimal factored forms for the class of RPO Boolean functions.

1.3 Thesis Organization

The remaining of this thesis is organized as follows.

Chapter 2: BOOLEAN LOGIC CONCEPTS — Provides to the reader basic and consolidated knowledge that is needed to understand the concepts presented in this work.

Chapter 3: READ-ONCE BOOLEAN FUNCTIONS — Describes the evolutionary line of RO factoring algorithms, and presents the two most recent methods that are related to the methods proposed herein.

Chapter 4: FACTORING INCOMPLETELY SPECIFIED BOOLEAN FUNCTIONS INTO READ-ONCE EQUATIONS — Presents a complete method for factoring ISBF into RO forms, when it is possible. This method, called ISF2RO could also be used in other applications, as discussed in Chapter 5.

Chapter 5: READ-POLARITY-ONCE FUNCTIONS — Defines a new class of Boolean functions, where each polarity (positive or negative) of a variable appears at maximum once in the minimum factored equation. Results of several experiments are also presented and discussed.

Chapter 6: CONCLUSIONS AND FUTURE WORK — Summarizes the major contributions of this work, and discusses some promising future work.


## 2 BOOLEAN LOGIC CONCEPTS

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

### 2.1 Sets, membership and basic operations

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

**Set**: a collection of distinct elements. The usual way of describing a set is by defining the characteristics of the elements belonging to the set. 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. This way, the set $A$ could be described as: $A = \{2, 4, 6, 8, 10, \ldots\}$.

**Membership**: If an element $a$ is member of the set $A$, this is denoted as $a \in A$.

One way of describing a set is by listing all elements of the set. One example of it is the definition of the binary set $B$ such that $B = \{0, 1\}$; meaning that $0 \in B$ and $1 \in B$.

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

With this very basic definition set, we can now provide several fundamental operations between sets, in order to provide new sets. Table 2.1 shows possible operations that can be performed over sets.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Venn Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Complement</strong>: If a universe $U$ is defined, $A^c$ creates a new set that contains all elements of a universe $U$ that are not in $A$: $U \setminus A$</td>
<td><img src="image" alt="Venn Diagram" /></td>
</tr>
<tr>
<td><strong>Union</strong>: $A \cup B$ creates a new set that contains all elements that are member of either $A$ or $B$.</td>
<td><img src="image" alt="Venn Diagram" /></td>
</tr>
<tr>
<td><strong>Intersection</strong>: $A \cap B$ creates a new set that contains all elements that are members of both $A$ and $B$.</td>
<td><img src="image" alt="Venn Diagram" /></td>
</tr>
<tr>
<td><strong>Symmetric difference</strong>: $A \Delta B$ creates a new set that contains all elements which are in either of the sets and not in their intersection, or more formally: $(A \cup B) \setminus (A \cap B)$</td>
<td><img src="image" alt="Venn Diagram" /></td>
</tr>
</tbody>
</table>
The goal of this review is just to present some very basic definitions and operations. For more information or formalism, (HALMOS, 1960) is suggested.

2.2 Boolean functions

Let $B = \{0,1\}$, $Y = \{0,1,X\}$. A Boolean function $f$ in $n$-input variables, $x_1,\ldots,x_n$, is a function:

$$f : B^n \rightarrow Y$$

where $x = [x_1,\ldots,x_n] \in B^n$ is the input of $f$. A **completely specified Boolean function** (CSF) $f$ is a logic function that allow only values $\{0,1\}$ in $Y$. Notice that **incompletely specified Boolean functions** (ISF) differs from completely specified functions in the fact that the former may also assume don’t-care ($X$) values, besides the binary values 0 and 1 (BRAYTON ET AL., 1984).

An element $m \in B^n$ is called term. The number of terms in $B^n$ is $2^n$. We can define the on-set $f^{ON}$ as the terms $m \in B^n$ such that $f(m) = 1$, the off-set $f^{OFF}$ such that $f(m) = 0$ and the don’t-care set $f^{DC}$, such that $f(m) = X$. The on-set and the off-set terms are also called minterms and maxterms, respectively. A CSF that contains no elements in the off-set ($f^{OFF} = \emptyset$) is defined as constant **ONE**. In analog way, a CSF that contains no elements in the on-set ($f^{ON} = \emptyset$) is defined as constant **ZERO**. In this work, Boolean function and function are used interchangeably.

2.2.1 Representing Boolean functions

There are several ways to represent a Boolean function. The most straightforward one is the truth table. In this representation, the value of the output is specified for all possible input vectors $[x_1,\ldots,x_n]$. For example, let $f: B^3 \rightarrow Y$ be specified by the truth table in Table 2.2. It is also possible to represent the same truth table in a bit string, that have the most significant bit on the left and the less significant bit on the right: $f(x1,x2,x3)=110010012$. A well-established way of representing Boolean functions is the **Karnaugh map** (KARNAUGH, 1953), and an example demonstrating its use is depicted in Figure 2.1.

<table>
<thead>
<tr>
<th>$x1$</th>
<th>$x2$</th>
<th>$x3$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Although being a very simple way of representing Boolean functions, the truth table data structure is not scalable in practice, since it uses $2^n$ bits to store the information. For functions with more than 20 input variables, representing Boolean functions as truth table data structure starts to be infeasible. In order to overcome this limitation, a graph-based approach has been proposed by (AKERS, 1978): Binary Decision Diagram (BDD).

A 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 and two children $low(v), high(v) \in V$. A terminal vertex $v$ has as attribute a value $v \in \{0,1\}$. The same function represented by the truth Table 2.2 above could be represented by a BDD as it is possible to see in Figure 2.2. The dashed edges represent the low child nodes, while the non-dashed lines are the high child nodes.

In order to make the BDD a canonical data structure, Bryant (BRYANT, 1986) proposed a reduced and ordered binary decision diagram (ROBDD). ROBDD is similar to the representation introduced by (AKERS, 1978), but with further restrictions on the ordering of decision variables in the graph. A ROBDD representing the same function of the BDD of Figure 2.2 could be seen in the Figure 2.3. Notice that ROBDD is also a more compact way of representing BDDs.
In general, BDDs represent only completely specified Boolean functions. In order to represent incompletely specified Boolean functions, two BDDs are necessary: one BDD to represent both on- and off-set, and another BDD for representing the don’t-care set of the function.

2.2.2 Operations on completely specified Boolean functions

In this subsection we define some basic Boolean operators for completely specified Boolean functions (CSF). After the basic operations definition, the next subsection defines how these operators work on incompletely specified Boolean functions (ISF).

The complement (NEGATION, NOT) of a CSF, \( f \) is a CSF, \( h = f^c \), \( (h = \neg f) \), such that \( h^\text{ON} = f^\text{OFF} \), and the \( h^\text{OFF} = f^\text{ON} \). A truth table representation of the NOT operator is presented in Table 2.3.

Table 2.3: Truth table representation of NOT operator.

<table>
<thead>
<tr>
<th>( f )</th>
<th>( \neg f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The union (SUM, OR) of two CSF, \( f \) and \( g \) is a CSF, \( h = f \cup g \), \( (h = f + g) \), such that \( h^\text{ON} \) is the union \( f^\text{ON} \) and \( g^\text{ON} \). A truth table representation of the OR operator is presented in Table 2.4.

Table 2.4 Truth table representation of OR operator.

<table>
<thead>
<tr>
<th>( f )</th>
<th>( g )</th>
<th>( f + g )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The intersection (PRODUCT, AND) of two CSF, \( f \) and \( g \) is a CSF, \( h = f \cap g \), \( (h = f \ast g) \), such that \( h^\text{ON} \) is composed by the terms that are in both \( f^\text{ON} \) and \( g^\text{ON} \). A truth table representation of the AND operator is presented in Table 2.5.

Table 2.5 Truth table representation of AND operator.

<table>
<thead>
<tr>
<th>( f )</th>
<th>( g )</th>
<th>( f \ast g )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The symmetric difference (exclusive-or, XOR) between two CSF, \( f \) and \( g \) is a CSF, \( h = f \triangle g \), \( (h = f \oplus g) \), such \( h^\text{ON} \) contains all elements which are in either \( f^\text{ON} \) or \( g^\text{ON} \) but not in their intersection. A truth table representation of the XOR operator is presented in Table 2.6.

Table 2.6 Truth table representation of XOR operator.

<table>
<thead>
<tr>
<th>( f )</th>
<th>( g )</th>
<th>( f \oplus g )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 2.6 Truth table representation of XOR operator.

<table>
<thead>
<tr>
<th>f</th>
<th>g</th>
<th>h = f ⊕ g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

We say that two CSF \( f \) and \( g \) are equal when \( f \equiv g \). In this case, \( f^{\text{ON}} \) and \( g^{\text{ON}} \) must be the same. We define equality between two CSF \( f \) and \( g \) as follows:

\[
f \equiv g \iff f^{\text{ON}} \cup g^{\text{ON}} = g^{\text{ON}}.
\]

2.2.3 Operations on incompletely specified Boolean functions

In the following the operations for incompletely specified Boolean functions (ISF) are defined. In an analog way, operations over ISF domain are quite related to CSF, regardless when a don’t-care term appears. A don’t-care term is a term that we indeed do not care about its value. In this sense, the following operations on ISF are defined: complement, union, intersection and symmetric difference.

The complement (NEGATION, NOT) of an ISF, \( f \) is a ISF, \( h = f^c \), (\( h = !f \)) defined as the following:

\[
h^{\text{ON}} = f^{\text{OFF}} \tag{1.1}
\]
\[
h^{\text{OFF}} = f^{\text{ON}} \tag{1.2}
\]
\[
h^{\text{DC}} = f^{\text{DC}} \tag{1.3}
\]

The union (SUM, OR) of two ISF, \( f \) and \( g \) is an ISF, \( h = f \cup g \), (\( h = f + g \)) defined as the following:

\[
h^{\text{ON}} = f^{\text{ON}} \cup g^{\text{ON}} \tag{1.4}
\]
\[
h^{\text{OFF}} = f^{\text{OFF}} \cap g^{\text{OFF}} \tag{1.5}
\]
\[
h^{\text{DC}} = U \setminus h^{\text{ON}} \setminus h^{\text{OFF}} \tag{1.6}
\]

The intersection (PRODUCT, AND) of two ISF, \( f \) and \( g \) is an ISF, \( h = f \cap g \), (\( h = f \ast g \)) defined as the following:

\[
h^{\text{ON}} = f^{\text{ON}} \cap g^{\text{ON}} \tag{1.7}
\]
\[
h^{\text{OFF}} = f^{\text{OFF}} \cup g^{\text{OFF}} \tag{1.8}
\]
\[
h^{\text{DC}} = U \setminus h^{\text{ON}} \setminus h^{\text{OFF}} \tag{1.9}
\]

The symmetric difference (exclusive-or, XOR) between two ISF, \( f \) and \( g \) is an ISF, \( h = f \Delta g \), (\( h = f \oplus g \)) defined as the following:

\[
h^{\text{ON}} = (f^{\text{ON}} \cup g^{\text{ON}}) \setminus (f^{\text{ON}} \cap g^{\text{ON}}) \tag{1.10}
\]
\[
h^{\text{OFF}} = (f^{\text{OFF}} \cap g^{\text{OFF}}) \cup (f^{\text{ON}} \cap g^{\text{ON}}) \tag{1.11}
\]
\[ h^{DC} = f^{DC} \cup g^{DC} \]  

(1.12)

Table 2.7: Summary of all binary operations over ISF in truth table data structure.

<table>
<thead>
<tr>
<th>f</th>
<th>g</th>
<th>h = f + g</th>
<th>h = f * g</th>
<th>h = f \oplus g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2.7 summarizes all binary operations over ISF in a truth table data structure. Two ISF \( f \) and \( g \) are said equal when \( f \equiv g \). More precisely, \( (f^{ON} = g^{ON}) \), \( (f^{OFF} = g^{OFF}) \) and \( (f^{DC} = g^{DC}) \). However, when in an ISF domain, it is needed often to verify if an ISF \( f \) is equivalent to another ISF \( g \). **Equivalence** between two ISF is defined as follows:

\[ f \approx g \iff f^{ON} \cap g^{OFF} = \emptyset \text{ and } f^{OFF} \cap g^{ON} = \emptyset \]

In order to demonstrate the equivalence operator, let \( f=11101X1011101X102 \) and \( g=1110111011X010X02 \) be two ISF. As it is possible to see in Figure 2.4, there is no element in \( f^{ON} \) that is also in \( g^{OFF} \), as well as no element in \( f^{OFF} \) that also belongs to \( g^{ON} \). In this case we say that \( f \) is equivalent to \( g \) (\( f \approx g \)).

Figure 2.4: Karnaugh map representation of \( f=11101X1011101X102 \) (a) and \( g=1110111011X010X02 \) (b). Dashes mean don’t-care (X) terms.

2.2.4 Cofactor over Boolean functions

The cofactor operation is a very basic and significant operation over Boolean functions. Let us define cofactor as following.

Let \( f: \mathbb{B}^n \rightarrow \mathbb{B} \) be a Boolean function and \( x = [x_1, ..., x_n] \) the variables in support of \( f \). The **cofactor** of \( f \) with respect to \( x_i \) is denoted as \( f(x_1, ..., x_i \leftarrow c, ..., x_n) \) where \( c \in \{0, 1\} \) (BOOLE, 1854). It is also possible to define as **positive cofactor** the operation where a variable receive the Boolean constant 1. The opposite is defined as **negative cofactor**,.
and is when a variable receives the Boolean constant 0. For presentation sake, let 
\( f(x_1, \ldots, x_i=c, \ldots, x_n) = f(x_i=c) \).

It is not a simple task to enumerate all methods that take advantage of the cofactor operation. One of the most important examples is the Shannon expansion, where a function can be represented as a sum of two sub-functions of the original (SHANNON, 1948):

\[
f(x_1, \ldots, x_n) = !x_i * f(x_i=0) + x_i * f(x_i=1)
\]  
(1.13)

Let us take as example a Boolean function \( f \) represented by the logic equation (1.14). Let the equation (1.15) and (1.16) represent, respectively, the negative and positive cofactor of variable \( x_2 \) over \( f \). The expansion of \( f \) about \( x_2 \) is defined by equation (1.17). Notice that both equation (1.14) and (1.17) represent the same Boolean function.

\[
f = !x_1 !x_2 !x_3 + x_1 x_2 + x_2 x_3
\]  
(1.14)

\[
f(x_2=0) = !x_1 !x_3
\]  
(1.15)

\[
f(x_2=1) = x_1 + x_3
\]  
(1.16)

\[
f = !x_2(!x_1 !x_3) + x_2(x_1+x_3)
\]  
(1.17)

### 2.2.5 Unateness

The unateness behavior is an intrinsic characteristic of Boolean functions. It provides the behavior of each variable, as well as the behavior of the entire function. Let \( f \) be a Boolean function. The unateness behavior of a variable \( x_i \) in \( f \) can be obtained according to the following relations:

\[
\alpha = f(x_i=1)
\]  
(1.18)

\[
\beta = f(x_i=0)
\]  
(1.19)

\[
\gamma = \alpha + \beta
\]  
(1.20)

- **don't-care:** \( \alpha \equiv \beta \)  
  (1.21)

- **positive unate:** \( (\alpha \equiv \gamma) \land (\alpha \neq \beta) \)  
  (1.22)

- **negative unate:** \( (\beta \equiv \gamma) \land (\alpha \neq \beta) \)  
  (1.23)

- **binate:** \( (\alpha \neq \beta) \land (\alpha \neq \gamma) \land (\beta \neq \gamma) \)  
  (1.24)

**Definition:** We say that a Boolean function is unate if all its variables are either positive or negative unate. When all variables are positive (negative) unate, we say that the entire function is positive (negative) unate. In the case when at least one variable is binate, the Boolean function is considered binate.

**Example:** Let \( f \) be a Boolean function defined by the following equation:

\[
f = !ab!cd+!abc!d+!abcd+a!bc!d+a!bcd+abc!d+abcd
\]  
(1.25)

which can be also represented by the bit string \( f=11001100111000002 \). In order to discover the unateness behavior of the variable “\( a \)”, we need to perform the positive and negative cofactors with regard to the variable “\( a \)”, as presented in Eq. 1.26 and Eq. 1.27, respectively. By applying a bitwise OR between the cofactors, the function \( \gamma_a \) is obtained, as shown in Eq. 1.28. As it is possible to see, the positive and negative
cofactors are not equals, as well as the function \( \gamma \) differs from both cofactors. This way, we determine that the variable “\( a \)” has a binate behavior in \( f \).
\[
\alpha_a = f(a=1) = 1100110011001100_2 \quad (1.26)
\]
\[
\beta_a = f(a=0) = 1110000011000000_2 \quad (1.27)
\]
\[
\gamma_a = a + \beta = 1110110011101100_2 \quad (1.28)
\]
In a straightforward way, it is possible to discover the unateness behavior of the variable “\( b \)” in \( f \). The equations Eq. 1.29 and Eq. 1.30 represent the positive and negative cofactors of “\( b \)” in \( f \), respectively. Through the bitwise OR between the cofactors, the function \( \gamma_b \) is obtained, as presented in Eq. 1.31. As it is possible to see, the positive and negative cofactors are not equals. However, the function \( \gamma_b \) (bitwise OR between cofactors of “\( b \)” is equivalent to the positive cofactor of “\( b \)”.) This means that the variable “\( b \)” has a positive unate behavior in \( f \). The same process is applied to the variables “\( c \)” and “\( d \)”, which have a positive unate behavior in \( f \). For simplicity sake, this process was omitted, since it is straightforward from equations Eq. 1.26, 1.27 and 1.28.
\[
\alpha_b = f(b=1) = 1100110011101110_2 \quad (1.29)
\]
\[
\beta_b = f(b=0) = 1100110000000000_2 \quad (1.30)
\]
\[
\gamma_b = a + \beta = 1100110011101110_2 \quad (1.31)
\]
In mathematics, a monotonic function (or monotone function) is a function between ordered sets that preserves some given order (GRÄTZER, 1971). Monotonic functions are divided between monotonically increasing and monotonically decreasing functions. Unate Boolean functions are monotonic functions with Boolean domain and image, where positive (negative) unate Boolean functions are monotonically increasing (decreasing) functions. For more information, (GRÄTZER, 1971) is suggested.

Unate Boolean functions are of special interest in this work. Some characteristics and applications of the unate Boolean functions are presented in Chapter 3. An algorithm that split binate variables into two independent unate variables, in order to transform a binate function into an unate one, is presented in Chapter 5. This algorithm and the application of this task are also discussed.

### 2.3 Logic equations

It is possible to use the set of Boolean operators in order to generate more complex structures. One example of complex structure is the algebraic representation of a Boolean function, which is simply named equation herein.

The simplest equation represents a Boolean variable, e.g. \( f = a \). This equation has only one literal. A literal is a variable or its complement (e.g. \( a \) or \( \neg a \)). In general, a Boolean function can be represented by several different equations. Let us take as example the CSF denoted by the bit string: \( f(x_1, x_2, x_3) = 11001001_2 \). It is possible to represent this function in sum-of-products (SOP) or in product-of-sums (POS) forms, as it is possible to see in Figure 2.5a and Figure 2.5b, respectively. We say that a SOP is an irredundant sum-of-products (ISOP) when no literal can be deleted from the SOP without changing the Boolean function. For instance, the SOP \( f = \neg x_1 \neg x_2 \neg x_3 + \neg x_1 x_2 x_3 + \neg x_1 x_2 + x_2 x_3 \) represents the same Boolean function as the ISOP \( f = \neg x_1 \neg x_2 \neg x_3 + x_1 x_2 + x_2 x_3 \). Moreover, we can delete literals from the SOP until it reaches an ISOP, without
changing function. In an analog way, the same property occurs for POS, and is called irredundant product-of-sums (IPOS).

There are some methods that produce more complex equations, known as factoring algorithms. **Factoring** is the process of deriving a parenthesized algebraic equation, or factored form, representing a given logic function, usually provided initially in SOP or in POS. The main target of factoring algorithms is to find an equation with the minimal number of literals as possible. For instance, let the input of a factoring algorithm be the SOP equation given by \( f = \overline{x_1} \overline{x_2} \overline{x_3} + x_1 x_2 + x_2 x_3 \). This equation can be factored into the logically equivalent one with only 6 literals: \( f = \overline{x_2}(\overline{x_1} \overline{x_3}) + x_2(x_1+x_3) \), while the original has 7 literals. Figure 2.6 shows a logic tree of the factored equation.

![Figure 2.5](image1.png)

**Figure 2.5:** The same Boolean function represented by an equation in sum-of-products (a) and in product-of-sums form (b).

![Figure 2.6](image2.png)

**Figure 2.6:** Arbitrary Boolean function represented by a factored form.
3 READ-ONCE BOOLEAN FUNCTIONS

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), data bases (KANAGAL; LI; DESHPANDE, 2011) and digital circuit design (PEER; PINTER, 1995). In this chapter, the concept and applications of read-once Boolean functions will be presented.

3.1 Definition

A Boolean function is considered read-once (RO) if it can be represented in a factored form where each variable appears only once (GOLUMBIC; MINTZ; ROTICS, 2001).

For example, the Boolean function represented by

\[ f = x_1 x_2 + x_1 x_3 x_4 + x_1 x_3 x_5 \]

is a read-once function since it can be factored into

\[ f = x_1 (x_2 + x_3 (x_4 + x_5)) \]

where each variable appears only once (GOLUMBIC; CRAMA; HAMMER, 2009).

If a given function \( f \) can be factored in a RO equation, then all variables of \( f \) are either positive or negative unate. If a function has some binate variable, this variable will appear in two phases, contradicting the read-once definition, where each variable appears at most once (LEE; WANG, 2007). This is a necessary but not sufficient condition, since there are functions composed by only unate variables that cannot be factored in a RO equation. For example, the unate function \( f = x_1 x_2 + x_1 x_3 + x_2 x_3 \) has \( f = x_1 (x_2 + x_3) + x_2 x_3 \) as the minimal solution, in which variables \( x_2 \) and \( x_3 \) appear more than once.

3.2 Previous work

The class of read-once Boolean functions is known for a long time, and was first introduced by Hayes (HAYES, 1975) and was called fanout-free functions. The method proposed by Hayes suffers of high complexity, since the algorithm makes intensive calls to a procedure to perform equivalence checking of cofactors, which was originally an expensive computation.
Peer and Pinter have also proposed in (PEER; PINTER, 1995) 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 calling several times a procedure that converts a SOP to POS (and POS to SOP) form. This original routine requires a non-polynomial time to run, making the method very costly in runtime.

More recent work was proposed in order to overcome the limitations of the 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 work (GURVICH, 1991). Another recent work was proposed by Lee (LEE; WANG, 2007) and is based on the Hayes work. His method, called herein as JPHI, replaced the equivalence checking of cofactors by a property called disappearance, turning the algorithm feasible in polynomial time. Both IROF and JPHI methods factorize RO functions in polynomial time. From this point on, the focus will be only on these two more recent approaches.

3.3 IROF method

In this section the basic idea of Golumbic’s approach to factorize read-once functions will be presented. More specific details can be found in the paper (GOLUMBIC; MINTZ; ROTICS, 2001).

The IROF method receives a Boolean function $f$ in an irredundant sum-of-products (ISOP) form. The first step is to build an undirected graph $G = (V, E)$, where $V = \{v_0, \ldots, v_n\}$ represents the set of literals of the ISOP. An edge $(v_i, v_j) \in E$ exists if the literals $v_i$ and $v_j$ appear in the same product in the ISOP.

Let us take an ISOP $f = x_1 x_2 + x_1 x_3 x_4 + x_1 x_3 x_5$ as example. The first step of IROF algorithm is to transform the ISOP into a graph. The resulting graph is depicted in Figure 3.1.

After the graph generation task, the algorithm proceeds to the partition step. Let $op \in \{+, *\}$ be a Boolean operator. Given a graph $G$ and an operator $op$ starting with ‘+’, the $\text{coGraph}_\text{Rec}(G, op)$ step first verifies if $G$ contains only one node. In this trivial situation, the method stores the operator $op$ on the node and returns. If $G$ contains more than one node, the algorithm proceeds checking if $G$ is connected or not. If $G$ is connected, the algorithm complement the graph $G$ and swap the operator $op$ (e.g. if $op$ is ‘+’, $op$’ will be ‘*’), making a recursive call $\text{coGraph}_\text{Rec}(G', op')$, where $G'$ is the...
complement of graph $G$. However, if $G$ is disconnected, the algorithm selects connected components from $G$, and for each connected component $H$, make a recursive call $\text{coGraph}\_\text{Rec}(H', \text{op}')$. The pseudo algorithm is presented in Figure 3.2

\begin{verbatim}
coGraph\_Rec(G, op) {
    if (G contains only one node $v_i$) {
        return op, $v_i$;
    }
    if (G is connected)
        return partition(G', $o'$);
    H = connected components of G;
    foreach ($H_i \in H$)
        partition($H_i'$, $o'$);
}
\end{verbatim}

Figure 3.2: Pseudo algorithm for the \textit{coGraph\_Rec} step.

Let us apply the \textit{coGraph\_Rec} step in the graph $G$ represented by Figure 3.1. Since $G$ contains more than one node and is a connected graph, a complemented graph $G'$ is created (Figure 3.3), and a recursive call $\text{coGraph}\_\text{Rec}(G', \text{op}')$ is called. The method is called recursively until there are no elements to be processed. The figures Figure 3.1, Figure 3.3, Figure 3.4, Figure 3.5 and Figure 3.6 show a complete example of the IROF algorithm. The algorithm successfully recognized the \textit{read-once} equation $f = x_1 (x_2 + x_3(x_4 + x_5))$ after the entire flow.

Figure 3.3: Complement graph of Figure 3.1. In this step, the \textit{coGraph\_Rec} build the partial equation: $f = x_1 \cdot f_1(x_2, x_3, x_4, x_5)$.

Figure 3.4: Complement graph of Figure 3.3. The node $x_1$ was removed. In this step, the \textit{coGraph\_Rec} build the partial equation: $f_1 = x_2 + f_2(x_3, x_4, x_5)$. 
Figure 3.5: Complement graph of Figure 3.4. The node $x_2$ was removed. In this step, the $coGraph\_Rec$ build the partial equation: $f_2 = x_3 \ast f_3(x_4, x_5)$.

Figure 3.6: Complement graph of Figure 3.5. The node $x_3$ was removed. In this step, the $coGraph\_Rec$ build the equation: $f_3 = x_4 + x_5$.

The IROF algorithm was the first method to factorize RO functions in a polynomial time. The IROF algorithm can be implemented in a time complexity $O(n|f|)$, where $|f|$ denotes the length of the ISOP equation of a function $f$, and $n$ is the number of variables in $f$ (GOLUMBIC; MINTZ; ROTICS, 2001). Despite the efficiency of the IROF algorithm, it cannot be modified to deal with incompletely specified Boolean functions, since it depends on an ISOP as input. Moreover, if the entire function is not read-once, the IROF method is not able to recognize subfunctions that are read-once. The JPHI method is able to find read-once subfunctions, even if the input function were not read-once. Furthermore, it is possible to modify the JPHI method to accept incompletely specified Boolean functions as input. JPHI method will be described in the next section.

3.4 JPHI method

This section will introduce the factoring algorithm for RO functions proposed by Lee (LEE; WANG, 2007), called herein as John P. Hayes Improved (JPHI). The goal of this section is to give a brief introduction about the method, and more details can be found in the paper.

The JPHI method receives a Boolean function $f$ in an irredundant sum-of-products (ISOP) form. We will consider ISOPs where all variables are positive unate, without losing generality. If a function has a negative unate variable, it is possible to substitute a positive unate variable for this negative unate variable. For example, the function $g(x_1, x_2) = x_1 + !x_2$ can be viewed as $f(x_1, !x_2) = x_1 + x_2$. 
The JPHI method is highly based on cofactors. The basic idea is to tie variables when their cofactors are equal. Let us take as example the function \( f = x_1 x_2 x_3 x_4 + x_1 x_2 x_3 x_5 + x_4 x_6 + x_5 x_6 \). The negative cofactor list is presented in Figure 3.7.

\[
\begin{align*}
  f(x_1 = 0) &= x_4 x_6 + x_5 x_6 \\
  f(x_2 = 0) &= x_4 x_6 + x_5 x_6 \\
  f(x_3 = 0) &= x_4 x_6 + x_5 x_6 \\
  f(x_4 = 0) &= x_1 x_2 x_3 x_5 + x_5 x_6 \\
  f(x_5 = 0) &= x_1 x_2 x_3 x_4 + x_4 x_6 \\
  f(x_6 = 0) &= x_1 x_2 x_3 x_4 + x_1 x_2 x_3 x_5 
\end{align*}
\]

Figure 3.7: Equations of negative cofactors of \( f = x_1 x_2 x_3 x_4 + x_1 x_2 x_3 x_5 + x_4 x_6 + x_5 x_6 \).

As it is possible to see, the negative cofactors of variables \( x_1 \), \( x_2 \) and \( x_3 \) have the same equation. In this way, we can tie these variables through an AND operator, e.g. \( \varphi = x_1 x_2 x_3 \), since the equations are from negative cofactors (Figure 3.8). When positive cofactors are equal, then the variables are tied through OR operators. After replacing \( f \) by \( \varphi \), the algorithm continues dealing with \( \varphi \) until reaching a RO form.

![Partial decomposition after tying variables into \( \varphi = x_1 x_2 x_3 \).](image)

The JPHI method uses the disappearance property to accelerate the JPH’s method (HAYES, 1975). The disappearance property can be defined as follows: Let \( x_i \) and \( x_j \) be two distinct variables of \( f \) and \( c \) be a Boolean constant. If \( x_j \) disappears in the function \( f(x_i = c) \) and \( x_i \) disappears in the function \( f(x_j = c) \), then \( f(x_i = c) \equiv f(x_j = c) \). A formal proof is given in (LEE; WANG, 2007).

In order to explore the disappearance property, a matrix can be build. This matrix has rows and columns labeled by \( f(x_i = c) \) and \( x_j \). The first step to build the matrix is to list the cofactors. The matrix is then filled with element 1 or 0 in position \((f(x_i = c), x_j)\) according to whether \( x_j \) appears in \( f(x_i = c) \). If \( x_j \) appears \( f(x_i = c) \), 1 is filled into \((f(x_i = c), x_j)\); otherwise, 0 is filled into \((f(x_i = c), x_j)\). This matrix has 0s on the diagonal because \( x_i \) always disappears in \( f(x_i = c) \). Through the disappearance property it is not more necessary to perform equivalence checking over cofactors, which was the main bottle neck of the JPH approach. The matrix based on the Figure 3.7 is depicted in Figure 3.9.
In a straightforward way, the positive cofactor list is evaluated and a disappearance matrix for the positive cofactors is built. After examining the adjacency relations among variables, it is possible to recognize the function \( f = (x_1x_2x_3 + x_6)(x_4 + x_5) \). Figure 3.10 shows the iterations when recognizing the *read-once* function \( f = (x_1x_2x_3 + x_6)(x_4 + x_5) \).

Besides the efficiently approach for recognizing *read-once* functions, the JPHI algorithm can also find *read-once* subfunctions even if the input function was not a *read-once* one. This is possible since the disappearance property tie variables in subfunction regardless tying the entire set of variables. Consider the function \( k=(x_1x_2x_3+x_6)(x_4+x_5)(ab+bc+ac) \) which is not *read-once* due to the term \((ab+bc+ac)\). JPHI method will produce a partially *read-once* equation, as it is shown in a circuit description in Figure 3.11. Despite the efficiency, the IROF algorithm is not able to recognize subfunctions that are *read-once*. If the same function \( k \) were used as input of IROF algorithm, it will just return that \( k \) is not a *read-once* function.
The time complexity of the JPHI algorithm is $O(n^2K)$, where $K$ denotes the number of products in the ISOP equation of a Boolean function $f$, and $n$ is the number of variables in $f$. In spite of the fact that the JPHI algorithm has higher complexity than the IROF algorithm, it is possible to modify the JPHI method in order to factorize incompletely specified Boolean functions as *read-once* solutions.

### 3.5 IROF and JPHI comparison

We will now briefly compare the IROF (Golumbic; Mintz; Rotics, 2001) and JPHI (Lee; Wang, 2007) methods. Table 3.1 compare the runtime (in seconds) required to synthesize an arbitrary benchmark of 9 *read-once* functions. The table was compiled with results regarding papers by Golumbic (IROF) and Lee (JPHI).

Table 3.1: Comparison regarding runtime (in seconds) required to synthesize an arbitrary benchmark of 9 *read-once* functions.

<table>
<thead>
<tr>
<th>SOP</th>
<th>N **</th>
<th>JPH</th>
<th>IROF</th>
<th>JPHI</th>
</tr>
</thead>
<tbody>
<tr>
<td>l2_b10</td>
<td>10240</td>
<td>20</td>
<td>4</td>
<td>0.3</td>
</tr>
<tr>
<td>l4_b3</td>
<td>3072</td>
<td>24</td>
<td>6</td>
<td>0.1</td>
</tr>
<tr>
<td>l4_b6</td>
<td>7290</td>
<td>24</td>
<td>277</td>
<td>0.21</td>
</tr>
<tr>
<td>l6_b4</td>
<td>672</td>
<td>20</td>
<td>3</td>
<td>0.02</td>
</tr>
<tr>
<td>l6_b8a</td>
<td>132</td>
<td>52</td>
<td>&gt;1hr</td>
<td>0.02</td>
</tr>
<tr>
<td>l6_b8b</td>
<td>24192</td>
<td>52</td>
<td>&gt;1hr</td>
<td>0.74</td>
</tr>
<tr>
<td>l8_b5</td>
<td>3380</td>
<td>29</td>
<td>&gt;1hr</td>
<td>0.09</td>
</tr>
<tr>
<td>l10_b3</td>
<td>2160</td>
<td>30</td>
<td>&gt;1hr</td>
<td>0.07</td>
</tr>
<tr>
<td>l14_b3</td>
<td>6720</td>
<td>42</td>
<td>&gt;1hr</td>
<td>0.2</td>
</tr>
</tbody>
</table>


*SOP*: Number of literals in the SOP equation.

**N**: Number of variables in the SOP equation.
In order to create a fairest comparison, both methods were implemented and tested over 3503 read-once functions from GenLib44-6 benchmark (SENTOVICH et al., 1992). This benchmark has functions from 1 to 16 inputs. The functions were grouped regarding input count, and then factorized by IROF and JPHI algorithms. The total runtime (in ms) to factorize each group of functions is presented in Table 3.2. It is possible to see in the table that both algorithms are equivalent in runtime from functions up to 8 inputs. From 9 inputs onwards, the runtime of the JPHI algorithm starts to grow quickly than the IROF. This behavior was expected, since the time complexity of JPHI is greater than IROF algorithm. It is possible to see in Figure 3.12 the average (in ms) runtime to synthesize the benchmark of 3503 read-once functions, grouped by number of inputs. The platform was a Linux system on Intel Core i5 2400 processor with 2GB main memory.

Table 3.2: Comparison of runtime between IROF and JPHI to synthesize the benchmark of 3503 read-once functions, grouped by number of inputs.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Functions</th>
<th>IROF* (ms)</th>
<th>JPHI** (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>116</td>
<td>31</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>228</td>
<td>83</td>
<td>81</td>
</tr>
<tr>
<td>9</td>
<td>374</td>
<td>115</td>
<td>144</td>
</tr>
<tr>
<td>10</td>
<td>530</td>
<td>162</td>
<td>188</td>
</tr>
<tr>
<td>11</td>
<td>612</td>
<td>168</td>
<td>247</td>
</tr>
<tr>
<td>12</td>
<td>604</td>
<td>171</td>
<td>305</td>
</tr>
<tr>
<td>13</td>
<td>468</td>
<td>145</td>
<td>310</td>
</tr>
<tr>
<td>14</td>
<td>300</td>
<td>107</td>
<td>297</td>
</tr>
<tr>
<td>15</td>
<td>130</td>
<td>41</td>
<td>184</td>
</tr>
<tr>
<td>16</td>
<td>48</td>
<td>19</td>
<td>100</td>
</tr>
</tbody>
</table>

\[ \sum \begin{array}{c} 3,503 \\ 1,068 \\ 1,928 \end{array} \]

*In-house implementation of the IROF algorithm presented in (GOLUMBIC; MINTZ; ROTICS, 2001).

**In-house implementation of the JPHI algorithm presented in (LEE; WANG, 2007).
The IROF algorithm was the first method to factorize RO functions in a polynomial time. The IROF algorithm can be implemented in a time complexity $O(n|f|)$, where $|f|$ denotes the length of the ISOP equation of a function $f$, and $n$ is the number of variables in $f$ (GOLUMBIC, MINTZ, ROTICS, 2001). As mentioned before, the IROF algorithm cannot deal with incompletely specified Boolean functions. Moreover, if the entire function is not read-once, the IROF method is not able to recognize subfunctions that are read-once. The JPHI method is able to find read-once subfunctions, even if the input function were not read-once. Furthermore, it is possible to modify the JPHI method to accept incompletely specified Boolean functions as input. JPHI method will be described in the chapter 4. Table 3.3 summarizes behavior comparison between IROF and JPHI algorithm.

Table 3.3: behavior comparison between IROF and JPHI algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Time complexity</th>
<th>Fail fast</th>
<th>Partial RO</th>
<th>Work with ISF*</th>
</tr>
</thead>
<tbody>
<tr>
<td>IROF</td>
<td>$O(n</td>
<td>f</td>
<td>)$</td>
<td>Yes</td>
</tr>
<tr>
<td>JPHI</td>
<td>$O(n^2K)$</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

$n$: number of variables in the SOP equation.
$|f|$: number of literals in the SOP equation.
$K$: number of products in the SOP equation.
*ISF: Incompletely Specified Boolean Functions
4 FACTORING INCOMPLETELY SPECIFIED BOOLEAN FUNCTIONS INTO READ-ONCE EQUATIONS

Efficient algorithms exist to perform factoring of read-once formulas. Most of them readily discard functions containing binate variables, as a read-once function is always unate. This work will extend the approach proposed by (LEE; WANG, 2007). The original Lee’s approach is described in the section 3.4. The modification for treating incompletely specified functions is described in this chapter. Several approaches (HAYES, 1975; GURVICH, 1991; PEER; PINTER, 1995; LEE; WANG, 2007; GOLUMBIC; MINTZ; ROTICS, 2001) have been proposed to identifying read-once functions (RO). Golumbic described the current state-of-the-art algorithm. The IROF algorithm needs an irredundant-sum-of-product (ISOP) as input and produces read-once formulae when this is possible or reports a failure otherwise. Despite the efficiency of the IROF algorithm, it cannot be modified to deal with incomplete-specified functions (ISF), since it depends on an ISOP as input. This is the reason why the Lee’s approach (LEE; WANG, 2007) was chosen to be extended. The original approach proposed by Lee was described in the section 3.4. The modification for treating incompletely specified functions is described as follows.

Let \( B = \{0, 1\} \) and \( Y = \{0, 1, X\} \). Let \( f \) be an incompletely specified Boolean function (ISF) of \( n \)-input variables \( x_1, x_2, \ldots, x_n \):

\[
f : B^n \rightarrow Y
\]

where \( x = [x_1, x_2, \ldots, x_n] \in B^n \). Notice that ISF differs from completely specified functions (CSF) in the fact that the former may also assume don’t-care (X) values, besides the binary values 0 and 1.

Methods for factoring ISF have been proposed in the literature (YOSHIDA; FUJITA, 2011; MARTINS ET AL., 2010), but only the Exact Factor approach guarantees exactness in the result (YOSHIDA; FUJITA, 2011). However, the method can take more than 10 minutes to synthesize any equation with 12 literals, even for RO functions (considering the computing platform Linux system on AMD Athlon 64 X2 4400 processor with 2 GB main memory (YOSHIDA; FUJITA, 2011).

**Definition 4.1:** If an ISF has a RO representation, there is at least one proper assignment of the don’t-care values that transforms this ISF into a CSF which is trivially synthesized through RO algorithms.
Unfortunately, to identify such CSF is not a straightforward task. An ISF can present a huge number of CSF forms, and only the CSFs that lead to unate functions are of interest. Hence, it is necessary that this CSF represents an unate function. However, such condition is not sufficient because the CSF must be successfully synthesized by RO algorithms.

Let us take as example the ISF \( f \) represented by the Karnaugh map depicted in Figure 4.1 (the don't-care terms appear as dashes). As an initial strategy, it is possible to apply a two-level minimizer method, like the Quine-McCluskey (MCCLUSKEY, 1956), in order to transform this ISF into a CSF that can be factorized into a read-once equation. However, this is a wrong strategy. Applying a two-level minimizer in the ISF \( f \) leads us to the SOP:

\[ f = x_3 x_4 x_5 x_6 + x_0 x_1 x_3 + x_0 x_1 x_2 \]  
(4.1)

which is functional represented by the Karnaugh map presented in Figure 4.2. Unfortunately, this SOP equation cannot be represented by a read-once formula. A naïve approach can assign all don't-care terms to constant zero or even to constant one. Unfortunately, these two resulting functions cannot be represented by read-once equations. However, with a proper assignment of the don't-care terms, we can obtain another SOP equation:

\[ f = x_0 x_1 x_2 + x_0 x_1 x_3 + x_0 x_2 x_4 x_6 + x_0 x_3 x_4 x_6 + x_1 x_2 x_5 + x_1 x_3 x_5 + x_2 x_4 x_5 x_6 + x_3 x_4 x_5 x_6 \]  
(4.2)

which can be successfully factorized into a read-once equation presented in Eq. 4.3. The correct assignment of the don't-care terms is depicted in the Figure 4.3.

\[ f = (x_0 + x_5) \ast (x_3 + x_2) \ast (x_1 + x_6 x_4) \]  
(4.3)
This function was synthesized by the Quine-McCluskey’s method and cannot be represented by a read-once formula.

As it is possible to see, it is not an easy task assigning the don’t-care terms in order to obtain a read-once equation. For example, consider the following ISF:

\[ f = 11101 X 1011 X 00000 \]  

We can assign the two don’t-care (X) in four different ways, as seen in Table 4.1.
Table 4.1: Don’t-care assignments and results.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>ISOP</th>
<th>RO</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>$x_1 x_4 + x_2 x_3$</td>
<td>$x_1 x_4 + x_2 x_3$</td>
</tr>
<tr>
<td>01</td>
<td>$x_1 x_4 + x_2 x_3 + x_2 x_4$</td>
<td>Error</td>
</tr>
<tr>
<td>10</td>
<td>$x_1 x_4 + x_1 x_3 + x_2 x_3$</td>
<td>Error</td>
</tr>
<tr>
<td>11</td>
<td>$x_1 x_4 + x_1 x_3 + x_2 x_3 + x_2 x_4$</td>
<td>$(x_1 + x_2) \cdot (x_3 + x_4)$</td>
</tr>
</tbody>
</table>

All don’t-care assignments lead to unate functions, but only two of them result in RO equations. In this sense, a method to assign the don’t-care terms in a RO driven way, called ISF2RO, is proposed. The algorithm is based on the principle discussed in (LEE; WANG, 2007), that compares cofactors in order to group variables in a RO fashion way. The ISF2RO algorithm is presented in the follow section.

4.1 ISF2RO method

In this section some basic data structures used in ISF2RO method will be presented.

Definition 4.2: An assignment is a data structure used to represent the state when a variable $x_i$ was assigned to a Boolean constant $c$. An assignment is represented by a tuple $<x_i, c>$. This structure will be used for cofactoring functions.

Definition 4.3: A logic arrangement is a data structure used to store the information necessary to run the algorithm, i.e., the grouping state, as illustrated in Figure 4.4

```
LogicArrangement

String expression
Set<Assignment> on_spec
Set<Assignment> off_spec
ISF pos_cube_cof
ISF neg_cube_cof
```

Figure 4.4: LogicArrangement class diagram.

The algorithm, called ISF2RO, receives an ISF $f$ as input. Initially, consider that all variables in $f$ are positive unate, without losing generality. The method starts by inserting the input variables into the main list of logic arrangements. These logic arrangements are cofactored and compared to each other. Let $x_i$ be a variable in $f$. The logic arrangement $la_i$ of $x_i$ is defined as follows:

$$la_i.expression = "x_i" \quad (4.5)$$

$$la_i.on_spec = \{ (x_i = 1) \} \quad (4.6)$$

$$la_i.off_spec = \{ (x_i = 0) \} \quad (4.7)$$

$$la_i.pos_cube_cof = f(la_i.on_spec) \quad (4.8)$$

$$la_i.neg_cube_cof = f(la_i.off_spec) \quad (4.9)$$
The proposed pseudo-algorithm is shown in Figure 4.5. The method starts by filling the main list with logic arrangements representing the input variables of the function \( f \), as observed in the line 1 in Figure 4.5. The next step consists in finding two logic arrangements that have the same cube cofactor function.

In order to illustrate the idea behind the method, let \( f = 11101X10111X00000 \). After filling the main list with the input variables, the expected list is shown in Table 4.2.

### Table 4.2: Expected initial main list.

<table>
<thead>
<tr>
<th>LA</th>
<th>Exp.</th>
<th>On</th>
<th>Off</th>
<th>Positive cube cofactor</th>
<th>Negative cube cofactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>( la_1 )</td>
<td>( x_1 )</td>
<td>( x_1 = 1 )</td>
<td>( x_1 = 0 )</td>
<td>11101X10111X01X0</td>
<td>11X0000011X00000</td>
</tr>
<tr>
<td>( la_2 )</td>
<td>( x_2 )</td>
<td>( x_2 = 1 )</td>
<td>( x_2 = 0 )</td>
<td>11101111X0111X0</td>
<td>1X101X1000000000</td>
</tr>
<tr>
<td>( la_3 )</td>
<td>( x_3 )</td>
<td>( x_3 = 1 )</td>
<td>( x_3 = 0 )</td>
<td>111111X11110000</td>
<td>10101010X0X00000</td>
</tr>
<tr>
<td>( la_4 )</td>
<td>( x_4 )</td>
<td>( x_4 = 1 )</td>
<td>( x_4 = 0 )</td>
<td>111111111XX0000</td>
<td>1100X0011000000</td>
</tr>
</tbody>
</table>

In Table 4.2, it is possible to see that \( la_1 \) has the same positive cofactor as \( la_2 \). Similar situation occurs between \( la_3 \) and \( la_4 \). Then, according to the lines (8-10, 17-19) in Figure 4.5, these logic arrangements could be grouped through an OR operator. There are also logic arrangements with equivalents negative cofactors: \( la_1 \approx la_4 \) and \( la_2 \approx la_3 \). Such logic arrangements are grouped by an AND operator in accordance to the lines (12-14, 17-19) in Figure 4.5. In Table 4.3, it is possible to see the result of first iteration of the proposed algorithm.

### Table 4.3: Grouped logic arrangements after first iteration.

<table>
<thead>
<tr>
<th>LA</th>
<th>Exp.</th>
<th>On</th>
<th>Off</th>
<th>Positive cube cofactor</th>
<th>Negative cube cofactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>( la_5 )</td>
<td>( x_1 + x_2 )</td>
<td>( x_1 = 1 )</td>
<td>( {x_1 = 0; x_2 = 0} )</td>
<td>11101X101111101X0</td>
<td>00000000000000</td>
</tr>
<tr>
<td>( la_6 )</td>
<td>( x_3 + x_4 )</td>
<td>( X_3 = 1 )</td>
<td>( {x_3 = 0; x_4 = 0} )</td>
<td>1111111111XX0000</td>
<td>000000000000000</td>
</tr>
<tr>
<td>( la_7 )</td>
<td>( x_1 * x_4 )</td>
<td>( {x_1 = 1; x_4 = 1} )</td>
<td>( x_1 = 0 )</td>
<td>1111111111111111</td>
<td>1100X0011000000</td>
</tr>
<tr>
<td>( la_8 )</td>
<td>( x_2 * x_3 )</td>
<td>( {x_2 = 1; x_3 = 1} )</td>
<td>( x_2 = 0 )</td>
<td>1111111111111111</td>
<td>10101010X0X00000</td>
</tr>
</tbody>
</table>

The algorithm continues grouping logic arrangements whenever it is possible. The stopping criteria of the algorithm is defined when a RO equation is recognized, as it is possible to see in the line (26) in Figure 4.5.
Table 4.4: Grouped logic arrangements after the second iteration.

<table>
<thead>
<tr>
<th>Exp.</th>
<th>On</th>
<th>Off</th>
<th>Pos. cube cofactor</th>
<th>Neg. cube cofactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x_1 \cdot x_4) + (x_2 \cdot x_3)$</td>
<td>${x_1=1; x_4=1}$</td>
<td>${x_1=0; x_2=0}$</td>
<td>1111111111111111</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>$(x_1 + x_2) \cdot (x_3 + x_4)$</td>
<td>${x_1=1; x_3=1}$</td>
<td>${x_1=0; x_2=0}$</td>
<td>1111111111111111</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

Figure 4.5: ISF2RO pseudo-algorithm.

After the second iteration (Table 4.4), two new logic arrangements can be found with the following equations:

$$f = (x_1 \cdot x_4) + (x_2 \cdot x_3)$$  \hspace{1cm} (4.10)

$$f = (x_1 + x_2) \cdot (x_3 + x_4)$$  \hspace{1cm} (4.11)
Since both (Eq. 4.10) and (Eq. 4.11) are RO equations, the test in the line (26) in Figure 4.5 will return both solutions.

The worst case time complexity of ISF2RO algorithm is $O(2^n)$, where $n$ is the number of variables in $f$. However, our empirical results are encouraging. The worst runtime observed in our results for functions up to 16 inputs was 500 seconds. The runtime is still lower than Exact Factor (YOSHIDA; FUJITA, 2011), even being able of dealing with more input variables. The Exact Factor algorithm takes 600 seconds to factorize an equation with 12 literals, while ours ISF2RO algorithm finds minimal equations for functions of 16 literals in less than 500 seconds.

In order to show the time complexity of the algorithm, an ISCAS85 benchmark of functions was synthesized. The functions obtained from the benchmark (72423 in total) were grouped regarding input count, from 3 up to 16 input variables. The worst case runtime for each group is depicted in Figure 4.6. The vertical axis is presented in logarithm scale. This way, it is easy to observe that the algorithm has an exponential time complexity. However, for this benchmark of functions, the worst case runtime was 20 seconds to synthesize a function with 16 inputs. The runtime which ISF2RO algorithm takes to factorize a function in 16 literals is still lower than the runtime of the Exact Factor algorithm (YOSHIDA; FUJITA, 2011) while factoring functions with only 12 literals.

![Figure 4.6: Worst case runtime (in ms) to synthesize functions from ISCAS’85 benchmark grouped by input count.](image-url)
5 READ-POLARITY-ONCE FUNCTIONS

Efficient exact algorithms exist for a sub-class of functions known as read-once functions (GOLUMBIC; MINTZ; ROTICS, 2001; LEE; WANG, 2007). A Boolean function is considered read-once (RO) if it can be represented in a factored form where each variable appears only once, e.g. \( f = a \cdot (b + c \cdot (d + e)) \). The class of RO functions is of special interest in logic synthesis, since they are quite frequent in circuit applications (PEER; PINTER, 1995).

However, exact algorithms for RO functions present two important limitations: (1) they do not consider incompletely specified Boolean functions (ISF); and (2) they are not suitable for functions with binate variables. In order to overcome the first constraint, I proposed an algorithm (ISF2RO) to find RO formulas for ISF, whenever possible (section 4.1). With respect to the second limitation, I propose a domain transformation, called here as a unatization process, that splits existing binate variables into two independent unate variables. Such domain transformation leads to ISF, which can be factored efficiently by the ISF2RO algorithm (section 4.1).

The combination of both contributions gives exact factoring results for a novel broader class of functions called read-polarity-once (RPO) functions, where each polarity (positive or negative) of a variable appears at maximum once in the factored expression. For instance, RO algorithms fail when factoring \( f = !a \cdot b \cdot d + b \cdot c + a \cdot c \), since the variable \( a \) is binate. The proposed RPO algorithm can factorize such function into an exact expression \( f = (!a \cdot d + c) \cdot (a + b) \), which presents only 5 literals.

Moreover, a study was made about the occurrence of RPO functions in circuits considering some ISCAS'85 benchmarks (IWLS, 2005). Experimental results have shown that RPO functions are significantly more frequent than RO functions. The entire flow comprising the unatization of RPO functions and the factoring of ISF in RO expressions has been validated. The implementation was able to efficiently find optimal solutions of functions up to 8 binate variables (i.e., up to 16 literals).

The remainder of this chapter is organized as follows. Section 5.1 presents definitions for the RPO class. Section 5.2 presents the proposed domain transformation, i.e., the unitization process of RPO functions. In section 5.3 is presented the complete algorithm to perform the factoring of ISF in RPO equations. Experimental results are shown in section 5.4.

5.1 Definition

Definition 5.1: A Boolean function is called read-polarity-once (RPO) if each polarity (positive or negative) of a variable appears at maximum once in the minimum factored equation (CALLEGARO et al., 2012).
**Lemma 5.1:** A positive (negative) unate variable contributes with at least one positive (negative) literal in a factored form.

**Lemma 5.2:** A binate variable contributes with at least two literals (one positive and one negative) in a factored form.

**Theorem 5.1:** A function represented by an RPO equation is in minimum form, if each unate variable contributes with exactly one literal and each binate variable contributes with exactly two literals (one positive and one negative).

**Proof:** Straightforward by lemmas 5.1 and 5.2.

**Corollary:** The proposed algorithm gives optimal results in literal count for RPO functions as it uses at most one literal per unate variable and at most two literals (one positive and one negative) per binate variables, while generating an RPO expression.

**Definition 5.2:** The RPO class is a superset of the RO class.

**Corollary:** Every RO function is also a RPO function, while a RPO function is not necessarily a RO function. For instance, \( f = a*(b+c*(d+e)) \) is both RO and RPO function, while \( f = (a+b)*(!a+!b) \) is only a RPO function.

The class of RPO functions is slightly different from the class of RO functions. The RPO class contains binate functions as elements (of the class), while the RO class contains only unate functions. Figure 5.1 shows a comparison between the universe of RO and RPO functions.

**Figure 5.1:** Comparison between the universe of read-once and read-polarity-once functions.

According to definition 5.1, if a function can be factored in a RPO expression, each polarity (positive or negative) of a variable appears at maximum once in the factored equation. Thus, an interesting point of investigation is if it is possible to separate the positive and negative literals, and transform the function into an unate function. Another point, equally interesting, is if that resulting transformation could be treated successfully by RO factoring algorithms.

In this sense, I propose a domain transformation (unatization) that splits existing binate variables into two independent unate variables. This domain transformation leads to ISF, which can be factored efficiently by the algorithm proposed in chapter 4. The combination of both contributions provides exact results for the class of RPO functions.

**5.2 Unatization process**

The unatization process is a fundamental step when recognizing read-polarity-once functions. This step could be seen as the major contribution of this work. Without the unatization process, the factoring flow herein presented would not exist. The starting point to obtain optimal equations for read-polarity-once functions was the very basic
idea of the domain transformation, where binate variables were split into two independent unate variables. The idea behind this process is that, if an unate function could be represented in a read-once form, this form will result in minimal literal count. By returning to an original domain of variables name, the resulting form will also still in optimal form regarding literal count. This is correct since each unate variable will appears once in the equation, while binate variables will appear twice (one literal to the positive and another one to the negative polarity).

Although been a critical step when recognizing read-polarity-once functions, it is important to notice that the unatization process can also be applied in other logic synthesis methods. It is possible to apply the unatization process on arbitrary Boolean functions in order to take advantage of methods that are specific designed for unate functions, e.g. unate recursive paradigm on ESPRESSO (BRAYTON, 1987). Another application example is the functional decomposition methods (BERTACCO; DAMIANI, 1997; MISHCHENKO; STEINBACH; PERKOWSKI, 2001), where the idea of splitting binate variables could result in better decomposed circuits.

**Definition 5.3:** The unatization method receives as input an ISF and split all binate variables into two independent unate variables.

Akers (AKERS, 1961) proposed a similar domain transformation that modifies a Boolean function into a logically passive function. However, the transformation proposed by Akers is devoted to find a single SOP representation (out of the many possible SOP representations). According definition 4.1, the specific SOP representation has to be chosen wisely to derive a RO expression. Our transformation into ISF exploits a broader space of solutions, where a given completely specified function satisfying the ISF can lead to a RO function (while others cannot).

**Definition 5.4:** Let \( f \) be an ISF, where \( f^{ON} \) represents the on-set of \( f \) and \( f^{DC} \) represents the don’t-care set of \( f \).

The pseudo-algorithm for the unatization process is shown in Figure 5.2. The basic idea behind the process is to split the binate variables into independent unate variables. Let \( x_i \) be a binate variable of \( f \). In order to unatize \( x_i \), a variable \( \text{not}_x_i \) is inserted into \( f \), as shown in the line 4 in Figure 5.2. It is important to notice that both variables \( x_i \) and \( \text{not}_x_i \) cannot have the same value at the same time. When a Boolean constant \( c \) is assigned to input \( x_i \), the complemented value has to be assigned to input \( \text{not}_x_i \). In this sense, the lines in the truth table where both variables are assigned to the same constant are set to don’t-care (see lines 5-6 in Figure 5.2), as these lines represent impossible input conditions. The next step is to guarantee that both variables \( (x_i \text{ and } \text{not}_x_i) \) became positive unate after the domain transformation.

**Table 5.1:** Two states that must be fixed in the unatization process.

<table>
<thead>
<tr>
<th>( f(x_i=1) )</th>
<th>( f(x_i=0) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

The \text{fix_positive_unate} method is based on the definition of unateness shown in subsection 2.2.5. Let \( x_i \) be the variable to be fixed, \( Y=\{0,1,X\} \) and \( \{t_p,t_m\} \in Y \) be a term
in the line $i$ of the truth table when $f(x_i=1)$ and $f(x_i=0)$, respectively (Table 5.1). The following two states must be fixed:

$$tp_i=0 \text{ and } tn_i = X$$  \hspace{1cm} (5.1)

$$tp_i=X \text{ and } tn_i = 1$$  \hspace{1cm} (5.2)

In Equation (5.1), if $tn_i$ receives the value $1$ the function becomes binate. The same happens in Equation (5.2) if $tp_i$ receives the value $0$. In order to avoid both situations, the method \texttt{fix_positive_unate} (see lines 14 to 25 in Figure 5.2) properly assign values to the don't-care terms that are responsible for these cases.

```
1  unatization($f^{ON}$, $f^{DC}$) {
2      for (int i = 0; i < n; i++){
3          if (is_binate(xi)){
4              createVariable(not_xi);
5              XNOR = !( xi ^ not_xi);
6              $f^{DC} =$ $f^{DC}$ + XNOR;
7              \texttt{fix_positive_unate($f^{ON}$, $f^{DC}$, xi)};
8              \texttt{fix_positive_unate($f^{ON}$, $f^{DC}$, not_xi)};
9          }
10      }
11  }
12
13  \texttt{fix_positive_unate($f^{ON}$, $f^{DC}$, xi)} {
14      PD = positive_cofactor($f^{DC}$, xi);
15      ND = negative_cofactor($f^{DC}$, xi);
16      PC = positive_cofactor($f^{ON}$, xi);
17      NC = negative_cofactor($f^{ON}$, xi);
18      state_0x = !PD * !PC * ND;
19      $f^{DC} =$ $f^{DC}$ * !state_0x;
20      $f^{ON} =$ $f^{ON}$ * !state_0x;
21      state_x1 = PD * !ND * NC;
22      $f^{DC} =$ $f^{DC}$ * state_x1;
23      $f^{ON} =$ $f^{ON}$ + state_x1;
24  }
```

Figure 5.2: Pseudo-algorithm for the unatization process.

The last situation that must be observed is when both $tp_i$ and $tn_i$ carry the value $X$. By definition, this situation does not turn the variable $x_i$ to a binate behavior. However, depending on the assignments of both don't-care terms, the behavior of the variable $x_i$ can be transformed into a binate function. This assignment is quite uncommon in the ISF2RO algorithm. When such case occurs, the logic arrangement cannot be grouped anymore. In order to avoid these cases, sets of pairs of functions to be distinguished (SPFD) could be considered to speed-up the algorithm (YAMASHITA; SAWADA; NAGOYA, 2000).

**Example:** Let $a$ and $b$ be binate variables from $f=(a+b)*(\neg a+\neg b)$. In order to unatize $f$, we introduce independent variables to represent the negative unate literals. Hence, by introducing variables $na$ and $nb$, a domain transformation is performed and the function...
becomes a 4-input function, with most of the terms appearing as *don’t-cares*, as seen in Figure 5.3. This is the first step of the unatization process.

The function represented by the *truth table* shown in Figure 5.3 is not positive unate. The second step of the unatization process is to turn all the variables to positive unate ones. Let us take as example the variable *a*. Figure 5.4 shows positive and negative cofactors of variable *a*. The lines in the *truth table* where the squares appear means that these lines must be fixed (Eq. 5.1 and Eq. 5.2) in order to turn the variable *a* in positive unate.

By computing the cofactors of the function and setting the *don’t-care* values to force the function to become positive unate in all of its variables, a new function is obtained. The computation of the cofactors and the new function obtained is presented in Figure 5.5.
Notice that two unspecified lines remain unspecified after this process (dashed rectangles presented in Figure 5.5). This is related to the unate flexibility of the function. Any of the four possible ways to assign these two lines will remain unate functions.

It is also worth to mention that the algorithm does not depend on the truth table data structure. In fact, our algorithm is based on reduced and ordered binary decision diagram (ROBDD) data structure (BRYANT, 1986).

The unatization algorithm can be implemented in time complexity of $O(n \times |m|)$, where $n$ is the number of variables in $f$ and $|m|$ is the number of nodes in the final ROBDD. Empirical results have shown that the unatization runtime is irrelevant in the entire flow for synthesize RPO functions. The runtime of the $ISF2RO$ algorithm is currently the main bottleneck.

### 5.3 Factoring incompletely specified boolean functions into read-polarity-once equations

Finally, it is possible to describe the entire flow of the RPO factoring algorithm, called $ISF2RPO$. The complete algorithm proceeds in two main steps. The first step reads an ISF $f$ and computes the polarity of the variables. Every binate variable is split into two separate positive unate variables according to the unatization process (section 5.2).

The ISF function returned by the unatization process is then used in the second step, where the $ISF2RO$ algorithm (section 4.1) performs a search for a RO equation. If $ISF2RO$ returns a RO output, the equation is then rewritten considering the original variables as they were presented before the domain transformation. An entire flow for recognize read-polarity-once functions is described in Figure 5.6.
Example: Let the input of the ISF2RPO algorithm be \( f = a \neg b + \neg a \ b \). The first step of the ISF2RPO is to split binate variables into unate ones through the unatization process (section 5.2). The ISF \( g \) obtained after the unatization process is presented in Figure 5.7. Notice that the negative polarities \( \neg a \) and \( \neg b \) are renamed to \( na \) and \( nb \), respectively.

\[
\begin{array}{ccc|c}
  \text{a} & \text{na} & \text{b} & \text{nb} & g \\
  0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 \\
  0 & 1 & 1 & 1 & x \\
  0 & 1 & 0 & 0 & 0 \\
  0 & 1 & 1 & 1 & 1 \\
  0 & 1 & 1 & 1 & 1 \\
  1 & 0 & 0 & 0 & 0 \\
  1 & 0 & 0 & 1 & 1 \\
  1 & 0 & 1 & 0 & 0 \\
  1 & 0 & 1 & 1 & 1 \\
  1 & 1 & 0 & 0 & x \\
  1 & 1 & 0 & 1 & 1 \\
  1 & 1 & 1 & 0 & 1 \\
  1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Figure 5.7: The ISF \( g \) obtained after the unatization process of the function \( f = a \neg b + \neg a \ b \).

After the unatization process, the function \( g \) is then used as input of the ISF2RO algorithm presented in section 4.1. Notice that two unspecified lines remain unspecified after the unatization process. This is directly related to the following two different equations obtained by the ISF2RO method: (Eq. 5.3) and (Eq. 5.4). Figure 5.8 shows how ISF2RO method set the remaining don t-care terms to produce different equations. Notice that both equations are in the minimal form, e.g. in a RO form.

\[
g = na \ast b + a \ast nb \tag{5.3}
\]

\[
g = (na + nb) \ast (a + b) \tag{5.4}
\]
The last step of the ISF2RPO flow is to rename variables with their original names. The equations Eq. 5.3 and Eq. 5.4 are renamed and are presented in Eq. 5.5 and Eq. 5.6, respectively. Notice that both equations are in the minimal form. They are structurally distinct and represent the same Boolean function.

\[ f = (!a \ast b) + (a \ast !b) \]  
\[ f = (!a + !b) \ast (a + b) \]  

The time complexity of the ISF2RPO algorithm is bounded by the complexity of the ISF2RO algorithm (see Figure 5.6). Experimental results demonstrate that the RPO algorithm can efficiently find optimal solutions in the number of literals for functions up to 8 binate variables (i.e., up to 16 unate literals).

### 5.4 Experimental Results

This section presents a study about the occurrence of RPO functions over the set of 5-input NPN-class, as well as the occurrence of RPO functions over the ISCAS'85 benchmark circuits (IWLS, 2005). The last experiment was carried out over a benchmark of important functions for logic brick design (MOTIANI; KHETERPAL; PILEGGI, 2010).

#### 5.4.1 NPN equivalence class up to 5 inputs

The first experiment was carried out over the set of all 5-input NPN-class (negation-permutation-negation) functions, grouped in 616,125 classes by equivalence through input permutation, negation of its inputs and/or negation of the output (CORREIA; 2001). Instead of running the algorithm for all Boolean space of 5 variables \(2^{32}\) functions), the 5-input NPN-class benchmark was chosen. The benchmark can represent the functionality of all Boolean space of 5 variables in a more compact set, without losing generality. To run the algorithm for all the 616,125 functions four minutes of
execution time were needed. The worst case optimization was 800 ms and the average case was less than 1ms. The platform was a Linux system on Intel Core i5 2400 processor with 2GB main memory.

For the universe of the 5-input NPN-class functions, there are 1,462 functions that are classified as RPO, while only 21 functions are classified as RO. This means that there are approximately 70x more RPO functions than RO functions, for the set of 5-input functions. Our results demonstrate that the universe of RPO is quite broader than the universe of RO functions, for which many works have been devoted (HAYES, 1975; GURVICH, 1991; PEER; PINTER, 1995; GOLUMBIC; MINTZ; ROTICS, 2001; LEE; WANG, 2007).

Comparative results evaluating the efficiency of the proposed algorithm are shown in Table 5.2, considering the set of 1,462 RPO functions. Our algorithm presented better results in terms of number of literals than Quick Factor (QF) (SETOVICH et al., 1992), Good Factor (GF) (SETOVICH et al., 1992), ABC (BERKELEY, 2012) and X-Factor (MINTZ; GOLUMBIC, 2005) tools. The proposed algorithm is still slow when compared to existing factoring methods (QF and GF methods). However, it guarantees minimal factored forms in literal count (theorem 5.1). It is important to notice that the ISF2RPO algorithm (section 5.3) factorize only RPO functions, while the other methods are of general factoring purposes.

Table 5.2: Total number of literals obtained and runtime when factoring process of 1,462 RPO functions using different approaches.

<table>
<thead>
<tr>
<th></th>
<th>QF</th>
<th>GF</th>
<th>ABC</th>
<th>X-Factor*</th>
<th>ISF2RPO (this work)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Literals</strong></td>
<td>16,086</td>
<td>15,671</td>
<td>15,981</td>
<td>13,253</td>
<td>13,064</td>
</tr>
<tr>
<td><strong>Runtime</strong></td>
<td>1.9s</td>
<td>2.3s</td>
<td>2.0s</td>
<td>7.1s</td>
<td>5.7s</td>
</tr>
</tbody>
</table>

* Results of an in-house implementation of the X-Factor algorithm.

5.4.2 ISCAS'85 benchmark suite

A study about the occurrence of RPO functions over some ISCAS'85 benchmarks (IWLS, 2005) was performed. Such analysis has been carried out in order to figure out the frequency of RPO functions in comparison to RO functions in mapped circuits. We have extracted functions up to 8 inputs from the benchmarks through K-Cuts method (MARTINELLO, 2010). These functions were grouped in P-classes, by equivalence through input permutations. The term ‘occurrences’ represents the number of functions counted before grouping them into P-classes of equivalence.
Table 5.3: Analysis of functions up to 8 inputs identified in ISCAS’85 benchmarks.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>RO</th>
<th></th>
<th>RPO</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P Classes</td>
<td>Occurrences</td>
<td>P Classes</td>
<td>Occurrences</td>
</tr>
<tr>
<td>2</td>
<td>67%</td>
<td>84%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>3</td>
<td>53%</td>
<td>66%</td>
<td>90%</td>
<td>88%</td>
</tr>
<tr>
<td>4</td>
<td>44%</td>
<td>54%</td>
<td>85%</td>
<td>71%</td>
</tr>
<tr>
<td>5</td>
<td>37%</td>
<td>42%</td>
<td>69%</td>
<td>53%</td>
</tr>
<tr>
<td>6</td>
<td>33%</td>
<td>36%</td>
<td>57%</td>
<td>46%</td>
</tr>
<tr>
<td>7</td>
<td>34%</td>
<td>36%</td>
<td>52%</td>
<td>47%</td>
</tr>
<tr>
<td>8</td>
<td>32%</td>
<td>34%</td>
<td>46%</td>
<td>44%</td>
</tr>
</tbody>
</table>

We have extracted the functions in two ways. Table 5.3 summarizes the results regarding all possible functions up to 8 inputs in the circuits. In Table 5.4 the functions were extracted from the circuits using an AIG greedy covering algorithm (MARTINELLO, 2010).

Similarly to RO functions, the number of RPO functions decreases as the number of variables increases. This is an expected result, since the more inputs the Boolean function have, the more complex the function become.

Table 5.4: Functions selected from the ISCAS’85 benchmarks using a greedy covering algorithm.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>RO</th>
<th></th>
<th>RPO</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P Classes</td>
<td>Occurrences</td>
<td>P Classes</td>
<td>Occurrences</td>
</tr>
<tr>
<td>2</td>
<td>78%</td>
<td>93%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>3</td>
<td>59%</td>
<td>63%</td>
<td>87%</td>
<td>94%</td>
</tr>
<tr>
<td>4</td>
<td>49%</td>
<td>53%</td>
<td>78%</td>
<td>81%</td>
</tr>
<tr>
<td>5</td>
<td>35%</td>
<td>36%</td>
<td>79%</td>
<td>81%</td>
</tr>
<tr>
<td>6</td>
<td>41%</td>
<td>39%</td>
<td>63%</td>
<td>60%</td>
</tr>
<tr>
<td>7</td>
<td>45%</td>
<td>43%</td>
<td>62%</td>
<td>57%</td>
</tr>
<tr>
<td>8</td>
<td>14%</td>
<td>15%</td>
<td>27%</td>
<td>27%</td>
</tr>
</tbody>
</table>

In Table 5.5, it is possible to see the runtime for synthesize all K-cuts functions up to K=8. In Table 5.6, it is shown the runtime of the algorithm for synthesizing the functions extracted from the covered circuit.
Table 5.5: Runtime for synthesizing all K-Cuts functions (K=8).

<table>
<thead>
<tr>
<th>Circuit</th>
<th>P Classes</th>
<th>Time (s)</th>
<th>Avg. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1355</td>
<td>680</td>
<td>123.455</td>
<td>0.18</td>
</tr>
<tr>
<td>C17</td>
<td>12</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>C1908</td>
<td>1224</td>
<td>133.162</td>
<td>0.11</td>
</tr>
<tr>
<td>C2670</td>
<td>6345</td>
<td>284.626</td>
<td>0.04</td>
</tr>
<tr>
<td>C3540</td>
<td>9275</td>
<td>123.945</td>
<td>0.01</td>
</tr>
<tr>
<td>C432</td>
<td>844</td>
<td>3.259</td>
<td>0.01</td>
</tr>
<tr>
<td>C499</td>
<td>432</td>
<td>98.187</td>
<td>0.23</td>
</tr>
<tr>
<td>C5315</td>
<td>11350</td>
<td>806.489</td>
<td>0.07</td>
</tr>
<tr>
<td>C6288</td>
<td>142</td>
<td>0.634</td>
<td>0.01</td>
</tr>
<tr>
<td>C7552</td>
<td>17888</td>
<td>8045.137</td>
<td>0.45</td>
</tr>
<tr>
<td>C880</td>
<td>1691</td>
<td>86.301</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Experimental results over ISCAS'85 benchmarks have demonstrated that RPO functions are significantly more frequent in circuit application than RO functions. The entire flow to factorize RPO functions was validated and our implementation was able to find optimal solutions of functions up to 8 binate variables in a reasonable runtime.

Table 5.6: Total runtime for synthesizing functions selected by a greedy covering algorithm.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>P Classes</th>
<th>Time (s)</th>
<th>Avg. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1355</td>
<td>16</td>
<td>3.70</td>
<td>0.23</td>
</tr>
<tr>
<td>C17</td>
<td>3</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>C1908</td>
<td>44</td>
<td>2.37</td>
<td>0.05</td>
</tr>
<tr>
<td>C2670</td>
<td>68</td>
<td>39.53</td>
<td>0.58</td>
</tr>
<tr>
<td>C3540</td>
<td>129</td>
<td>1.29</td>
<td>0.01</td>
</tr>
<tr>
<td>C432</td>
<td>25</td>
<td>0.15</td>
<td>0.01</td>
</tr>
<tr>
<td>C499</td>
<td>10</td>
<td>0.89</td>
<td>0.09</td>
</tr>
<tr>
<td>C5315</td>
<td>108</td>
<td>58.07</td>
<td>0.54</td>
</tr>
<tr>
<td>C6288</td>
<td>38</td>
<td>0.20</td>
<td>0.01</td>
</tr>
<tr>
<td>C7552</td>
<td>130</td>
<td>8.42</td>
<td>0.06</td>
</tr>
<tr>
<td>C880</td>
<td>36</td>
<td>18.08</td>
<td>0.50</td>
</tr>
</tbody>
</table>
According to (MOTIANI; KHETERPAL; PILEGGI, 2010), a set of logic functions can be added to a cell library to significantly improve specific designs. In one of the examples given by Motiani, a set of 12 distinct functions were added to the library. Out of the twelve functions added, 6 were RO, 10 were RPO (including the 6 RO that are also RPO) and only 2 are not RPO functions, as it is possible to see in Table 5.7. This observation highlights the importance of the RPO class for different technologies, including the logic brick methodology proposed by (MOTIANI; KHETERPAL; PILEGGI, 2010).

Table 5.7: Set of 12 distinct functions given by Motiani where 10 were RPO functions.

<table>
<thead>
<tr>
<th>Original</th>
<th>Read-polarity-once equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>f01 = p0p1p3+p2p3+p4p5</td>
<td>f01 = (((p1 * p0) + p2) * p3) + (p4 * p5)</td>
</tr>
<tr>
<td>f02 = p0p1p3+p3p4+p1p2p3</td>
<td>f02 = (((p2 + p0) * p1) + p4) * p3)</td>
</tr>
<tr>
<td>f03 = p1p3p6+p0p2p5+p3p4p6+p1p2p4</td>
<td>f03 = (p4 * p1 + p0 * p5) * p2 + p6 * p3 * (p4 + p1)</td>
</tr>
<tr>
<td>f04 = p1p2p3+p0p1p3+p1p3+p0p2p3</td>
<td>f04 = (((p2 * p0) + p1) * p3) + (p1 * p3) * p0 + p2))</td>
</tr>
<tr>
<td>f05 = p1p4+p0p3+p1p3+p2p4+p0p4+p2p3</td>
<td>f05 = (((p0 + p1) + p2) * (p3 + p4))</td>
</tr>
<tr>
<td>f06 = p0p2+p1p2+p3p4</td>
<td>f06 = (((p1 + p0) * p2) + (p4 * p3))</td>
</tr>
<tr>
<td>f07 = p1p2+p4p5+p6p7+p0p3</td>
<td>f07 = (((p1) * p2) + (p3) * (p0)) + (p6 * p7) + (p4 * p5))</td>
</tr>
<tr>
<td>f08 = p1p3p4+p0p2p3p4+p1p2p3p4+p0p1p3p4+p1p2p3p4+p1p3p4+p0p1p2p4</td>
<td>f08 = Non RPO</td>
</tr>
<tr>
<td>f09 = p0p3p5+p0p1p2+p2p3p4+p1p4p5+p1p3p5+p0p4p5</td>
<td>f09 = (p4 + p3 + p1 + p0) * p2 + (p3 + p4) * p5 * (p0 + p1)</td>
</tr>
<tr>
<td>f10 = p0p1p2+p0p1p3+p0p1p2+p0p1p3</td>
<td>f10 = Non RPO</td>
</tr>
<tr>
<td>f11 = p1p2p3+p0p3+p1p2p4+p0p4</td>
<td>f11 = (((p1 * p2) + p0) * (p4 + p3))</td>
</tr>
<tr>
<td>f12 = p0p1p4+p2p4+p0p2p3+p1p2p3</td>
<td>f12 = (((p1 + p0) * p2) + p4) * (p1 + p0) * (p3 + p2))</td>
</tr>
</tbody>
</table>
CONCLUSIONS AND FUTURE WORK

The main contribution of this work was the introduction of the concept of read-polarity-once functions. Besides introducing the class of RPO functions, several related contributions were also introduced: (1) an algorithm for factoring incompletely specified functions into read-once equations; (2) a domain transformation that splits existing binate variables into two independent unate variables; and (3) a complete algorithm for exact factoring the class of read-polarity-once functions.

The first contribution was the proposal of an algorithm (ISF2RO) that is able to factorize incompletely specified Boolean functions into read-once forms, when this is possible. To the best of the author’s knowledge, the unique method that guarantees exactness in the result is the Exact Factor approach, proposed by (YOSHIDA; FUJITA, 2011). However, the method is feasible with functions with up to 10 variables. The ISF2RO proposed in this work was tested and the results show that it is able to factorize functions with up to 16 input variables.

The second contribution was the proposal of a method for domain transformation that splits existing binate variables into two independent unate variables, called unatization process. The unatization process is a fundamental step when recognizing read-polarity-once functions. However, it can be applied in other logic synthesis methods, like functional decomposition methods.

The third contribution was the development of a complete algorithm for exact optimal factoring the class of read-polarity-once functions called ISF2RPO. Such algorithm was implemented and compared to existing factoring algorithms. The proposed algorithm guarantees minimal factored forms for the class of read-polarity-once functions. Results show that, out of the set of 612,125 NPN-class functions with up to 5-inputs, 1,462 functions were identified as read-polarity-once, while only 21 functions can be considered read-once. Moreover, results taking into account ISCAS'85 benchmarks have shown that read-polarity-once functions are quite more frequent in circuit application than read-once functions. The read-polarity-once class of functions is also important for different technologies, including the logic brick methodology proposed (MOTIANI; KHETERPAL; PILEGGI, 2010), as demonstrated by the large number of read-polarity-once functions (10 out of 12) given as example in the (United States Patent number 7784013).

It is worthy to highlight that both algorithms were implemented in a tool, called SwitchCraft (CALLEGARO et al., 2010). SwitchCraft framework provides a set of tools for switch network and logic gate generation. The tool has been applied at Federal University of ‘Rio Grande do Sul’ (UFRGS), in Brazil, in undergraduate and graduate courses about fundamentals of IC digital design.
There is still much work that needs to be carried on. First of all, the runtime of the ISF2RO must be improved. A new version of the algorithm based on branch-and-bound is already in development. Another point to be carried is with regard to the scalability of the ISF2RO method. The current version of the algorithm scales up to 16 variables. I believe that will be possible to scale to 32 variables with the new version of the algorithm.
7 REFERENCES


A.1 Introdução

O fluxo de projeto de síntese de circuitos é normalmente dividido em três etapas principais: síntese arquitetural, síntese lógica e síntese física (MICHELI, 1994). A síntese de arquitetural, muitas vezes chamada de síntese de alto nível, consiste em transformar uma descrição algorítmica de um comportamento desejado em um formato de hardware que implementa esse comportamento, como no formato RTL (Register Transfer Level). Normalmente, essas descrições algorítmicas são representadas em um formato semelhante à linguagem C, como (por exemplo, System C) ou formato HDL (Hardware Description Language) comportamental (por exemplo, VHDL e Verilog).

O processo de síntese lógica tem sido uma das áreas de maior sucesso comercial na automação de projetos eletrônicos (EDA). Este sucesso comercial indica que todos os dispositivos digitais que usamos no nosso dia-a-dia foram projetados com um conjunto de ferramentas de síntese lógica. A tarefa de síntese lógica é composta por várias etapas. Estas etapas podem ser diferentes de acordo com a natureza do circuito, por exemplo, como circuito sequencial ou de combinacional. O objetivo da síntese lógica é determinar a estrutura primitiva de um circuito, ou seja, a sua representação em nível de portas lógicas primitivas. A síntese lógica é normalmente dividida em três fases: otimizações independentes de tecnologia, mapeamento tecnológico e otimizações dependentes de tecnologia. A primeira aplicam-se transformações que não dependem da tecnologia, mas sim do comportamento funcional de uma rede Booleana, por exemplo, algoritmos de fatoração. Em seguida, a fase de mapeamento tecnológico combina porções do circuito para uma célula informação de tecnologia. A fase dependente de tecnologia aplica otimizações no circuito mapeado, tais como redimensionamento de células e duplicação lógica.

A síntese física consiste principalmente em duas tarefas principais: o posicionamento de blocos e roteamento dos fios que os conectam. O primeiro distribui fisicamente as células enquanto o posterior executa as interligações de sinal.

Este trabalho aborda a síntese de funções Booleanas no âmbito de um fluxo de projeto de circuito digital, mais precisamente na fase de síntese lógica. No entanto, o foco pode ser considerado mais amplo, já que este trabalho propõe uma nova classe de funções Booleanas que podem ter aplicação em diferentes outras áreas além da síntese de circuitos.

A.1.1 Motivação

O processo de fatoração de funções booleanas é uma operação fundamental na síntese lógica algorítmica (BRAYTON, 1987; HACHTHEL; SOMENZI, 2006). Fatoração é o processo de derivação de uma equação algébrica, ou forma fatorada, que
representa uma determinada função lógica, normalmente descrita inicialmente em forma de soma de produtos (SOP) ou produto-de-somas (POS). Por exemplo, \( f = a \cdot b + a \cdot c \cdot d + a \cdot c \cdot e \) pode ser fatorado em uma equação logicamente equivalente \( f = a \cdot (b + c \cdot (d + e)) \).

Qualquer função lógica pode ser representada por diferentes equações fatoradas. A tarefa de fatorar funções Booleanas fórmulas menores, mais compactas e logicamente equivalentes é uma das operações básicas nos estágios iniciais da síntese lógica algorítmica (HACHTEL; SOMENZI, 2006). Na maioria dos estilos de projeto, como portas lógicas CMOS convencional, o custo de implementação elétrica de uma função Booleana corresponde diretamente à sua equação fatorada em relação a número de literais e contagem de dispositivos (transistores). A geração de uma forma fatorada exata, ou seja, a equação de menor comprimento é um problema NP-difícil (GOLUMBIC; MINTZ, 1999). Assim, algoritmos heurísticos foram desenvolvidos a fim de obter boas soluções fatoradas (BRAYTON, 1987; STANION; SEHEN, 1994; MINTZ; GOLUMBIC, 2005; HACHTEL; SOMENZI, 2006; YOSHIDA; FUJITA, 2011). Alguns algoritmos heurísticos conhecidos incluem XFactor (MINTZ; GOLUMBIC, 2005), que proporciona bons resultados, mas não garante as equações mínimas. Em (LAWLER, 1964), o autor pretende fornecer um algoritmo para fatoração exata. No entanto, o método de Lawler não é escalável e torna-se impraticável mesmo para funções com apenas quatro variáveis. Recentemente, novas abordagens têm melhorado o processo de fatoração de soluções exatas, mas a escalabilidade e tempo de execução continuam a serem os principais gargalos (YOSHIDA; IKEDA; ASADA, 2006; YOSHIDA; Fujita, 2011; MARTINS ET AL, 2012).

Algoritmos exatos e eficientes existem para uma subclasse de funções Booleanas conhecidas como read-once (RO) (LEE; WANG, 2007; GOLUMBIC; MINTZ; ROTICS, 2008). Uma função Booleana é considerada RO se ela pode ser representada por uma forma fatorada onde cada variável apareça apenas uma vez (GOLUMBIC; MINTZ; ROTICS, 2001). Como no exemplo acima mencionado, a função \( f = a \cdot (b + c \cdot (d + e)) \) é RO.

No entanto, algoritmos exatos para funções RO apresentam duas limitações importantes: (1) eles não consideram funções booleanas incompletamente especificadas (ISBF), e (2) eles não são adequados para as funções com variáveis binate (ver Subseção 2.2.5). Neste contexto, uma questão pode ser levantada: “Uma vez que as funções \( RO \) são importantes em aplicações reais de circuitos, suponhamos que estendem os limites desta classe, criando uma nova classe de funções, a fim de ultrapassar a limitação (1) e (2). O quanto mais abrangente seria esta nova classe no que diz respeito à classe RO? Quanto mais importante serão as funções dessa nova classe em aplicações reais de circuitos?” Essas perguntas resumem a motivação desta dissertação.

A.1.2 Objetivos

O objetivo desta dissertação é a de transcender os limites da classe RO de funções. Uma maneira de fazer isso é criar uma nova e ampla classe de funções Booleanas, que pode lidar com funções binate e onde formas fatoradas são mínimas. Uma abordagem é a de dividir as fases positivas e negativas de variáveis binate em duas variáveis independentes. Desta forma, é possível representar a fase positiva em uma variável e a fase negativa em outra variável, sendo essas duas variáveis unate e independentes umas das outras. Esta transformação leva a um domínio de funções Booleanas
incompletamente especificadas (ISBF). Assim, é importante que se desenvolva um método que seja capaz de fatorar ISBF em formas RO, que são conhecidos por resultar em formas fatoradas mínimas. A combinação de ambas as estratégias podem levar a uma nova classe mais ampla de funções chamadas funções read-polarity-once (RPO), onde cada uma das polaridades (positiva ou negativa) de uma variável aparece no máximo uma vez na forma fatorada mínima, como por exemplo, \( f=(\overline{a}d+c)(a+b) \). Neste sentido, este trabalho tem como objetivo fornecer um algoritmo eficiente que garanta formas fatoradas mínimas para a classe de funções Booleanas RPO.
A.2 Funções Read-Polarity-Once

Existem algoritmos exatos para uma subclasse de funções conhecidas como funções read-once (GOLUMBIC; MINTZ; ROTICS, 2001; LEE; WANG, 2007). Uma função Booleana é considerada read-once (RO), se ela pode ser representada numa forma onde cada variável apareça uma única vez, por exemplo, \( f = a \cdot (b + c \cdot (d + e)) \). A classe de funções RO é de especial interesse na síntese lógica, uma vez que eles são bastante frequentes em aplicações de circuitos (PEER; PINTER, 1995).

No entanto, algoritmos exatos para funções RO apresentam duas limitações importantes: (1) eles não consideram funções Booleanas incompletamente especificadas (ISF), e (2) não são adequados para as funções com variáveis binate. A fim de superar a primeira restrição, um algoritmo (ISF2RO) será aqui proposto para encontrar fórmulas RO para ISF, sempre que possível. No que diz respeito à segunda limitação, uma transformação de domínio será aqui proposta, que será chamada como um processo unatization, que divide variáveis binate em duas variáveis unate independentes. Esta transformação leva a domínio ISF, que pode ser fatorada de forma eficiente pelo algoritmo ISF2RO proposto.

A combinação de ambas as contribuições dá resultados exatos para a fatoração de uma nova classe de funções chamadas de funções read-polarity-once (RPO), onde cada uma das polaridades (positiva ou negativa) de uma variável aparece no máximo uma vez na expressão. Por exemplo, os algoritmos para RO falham ao fatorar \( f = !a \cdot b \cdot d + b \cdot e + a \cdot c \), uma vez que a variável ‘\( a \)‘ é binate. O algoritmo RPO proposto pode fatorar essa função em uma expressão exata: \( f = (!a \cdot d + c) \cdot (a + b) \), que apresenta apenas 5 literais.

Além disso, um estudo foi feito sobre a ocorrência de funções RPO em circuitos considerando benchmarks ISCAS’85 (IWLS, 2005). Os resultados experimentais mostraram que funções de RPO são significativamente mais frequentes do que as funções RO. Todo o fluxo compreendendo o processo de unatization de funções RO e a fatoração de funções ISF em expressões RO foi validado. A implementação foi capaz de encontrar soluções exatas de forma eficiente para funções de até 8 variáveis binate (ou seja, até 16 literais).

A.2.1 Definição de funções Read-Polarity-Once

Definição 5.1: Uma função Booleana é chamada de read-polarity-once (RPO), se cada uma das polaridades (positiva ou negativa) de uma variável aparece, no máximo, uma vez na equação fatorada mínima (CALLEGARO et al, 2012).

Lema 5.1: uma variável unate positiva (negativa) contribui com, pelo menos, um literal positivo (negativo) em uma forma fatorada.

Lema 5.2: uma variável binate contribui com pelo menos dois literais (um positivo e um negativo) em uma forma fatorada.

Teorema 5.1: a função representada por uma expressão RPO está na forma mínima, se cada variável unate contribui com exatamente um literal e cada variável binate contribui com exatamente dois literais (um positivo e um negativo).

Prova: Direta dos lemas 5.1 e 5.2.

Corolário: O algoritmo proposto fornece resultados exatos na contagem de literais para funções RPO, pois ele usa no máximo um literal por variável unate e no máximo
dois literais (um positivo e um negativo) por variáveis binate, gerando uma expressão RPO.

**Definição 5.2:** A classe RO é um subconjunto da classe RPO.

**Corolário:** Cada função RO também é uma função RPO, enquanto que uma função RPO não é necessariamente uma função RO. Por exemplo, \( f = a*(b+c*(d+e)) \) é tanto uma função RO quanto RPO, enquanto que \( f = (a+b)*(!a+!b) \) é apenas uma função RPO.

A definição da classe de funções de RPO é ligeiramente diferente da classe de funções de RO. A classe RPO contém funções binate como elementos (da classe), enquanto que a classe RO contém apenas funções unate. A Figura A.1 mostra uma comparação entre o universo de funções RO e funções RPO.

Figura A.1: Comparação entre o universo de funções read-once e read-polarity-once.

De acordo com a Definição 5.1, se a função pode ser fatorada numa expressão RPO, cada polaridade (positiva ou negativa) de uma variável aparece, no máximo, uma vez na forma fatorada. Assim, um aspecto interessante para investigação é se é possível separar os literais positivos e negativos, e transformar a função a uma função unate. Outro ponto, igualmente interessante, é se essa transformação resultante pode ser tratada com sucesso por algoritmos de fatoração RO.

É possível descrever todo o fluxo do algoritmo de fatoração RPO, chamado ISF2RPO. O algoritmo completo procede em duas etapas principais. A primeira etapa lê um ISF f e calcula a polaridade das variáveis. Cada variável binate é dividida em duas variáveis unate positivas de acordo com o processo de unatization (seção 5.2).

A função ISF devolvida pelo processo unatization é então utilizado na segunda etapa, em que o algoritmo ISF2RO (seção 4.1) realiza a fatoração para uma equação RO. Se o algoritmo ISF2RO retorna uma saída RO, a equação é então reescrita considerando as variáveis originais que foram apresentadas antes a transformação de domínio. Um fluxo inteiro para reconhecer as funções read-polarity-once é descrito na Figura A.2.

Figura A.2: Fluxograma do algoritmo ISF2RPO.
A.2.2 Resultados experimentais

Esta seção apresenta um estudo sobre a ocorrência de funções RPO sobre o conjunto de funções de 5 entradas da classe NPN, bem como a ocorrência de funções RPO em circuitos ISCAS’85 (IWLS, 2005).

A.2.2.1 Classe de equivalência NPN de 5 entradas

O primeiro experimento foi realizado sobre o conjunto de todas as classes de funções NPN (negação - permutação - negação) de 5 entradas. Estas funções são agrupadas em 616.125 classes de equivalência através permutação de entradas, negação de entradas e / ou negação de saída (CORREIA; 2001). O benchmark pode representar a funcionalidade de todo o espaço Booleano de cinco variáveis (2^{32} funções) em um conjunto mais compacto, sem perder generalidade. Para executar o algoritmo para todas as 616.125 funções foram necessários quatro minutos de tempo de execução. O pior caso foi a otimização de 800ms e o caso médio foi inferior a 1ms. A plataforma de execução é um sistema Linux em processador Intel Core i5 2400 com 2GB de memória principal.

Para o universo das funções NPN de 5 entradas, existem 1.462 funções que são classificadas como RPO, enquanto que apenas 21 funções são classificadas como RO. Isto significa que existem cerca de 70 vezes mais funções RPO do que funções RO, para o universo das funções NPN de 5 entradas. Nossos resultados demonstram que o universo da RPO é muito mais amplo do que o universo de funções RO, ao quais muitas obras foram dedicadas (HAYES, 1975; GURVICH, 1991; PEER; PINTER, 1995; GOLUMBIC; MINTZ; ROTICS, 2001; LEE; WANG, 2007).

Os resultados comparativos obtidos avaliando a eficiência do algoritmo são apresentados na Tabela A.2.1, considerando o conjunto as 1.462 funções RPO. Nosso algoritmo apresentou melhores resultados em termos de número de literais que as ferramentas Quick Factor (QF) (SENTOVICH et al., 1992), Good Factor (GF) (SENTOVICH et al., 1992), ABC (BERKELEY, 2012) and X-Factor (MINTZ; GOLUMBIC, 2005). O algoritmo proposto ainda é lento quando comparado aos métodos existentes de fatoração (QF e GF). No entanto, ele garante formas fatoradas mínimas em relação à contagem de literais (Teorema 5.1).

Tabela A.2.1: Número total de literais obtidos e tempo de execução, ao fatorar 1.462 funções RPO considerando diversas abordagens.

<table>
<thead>
<tr>
<th></th>
<th>QF</th>
<th>GF</th>
<th>ABC</th>
<th>X-Factor*</th>
<th>ISF2RPO (este trabalho)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literais</td>
<td>16,086</td>
<td>15,671</td>
<td>15,981</td>
<td>13,253</td>
<td>13,064</td>
</tr>
<tr>
<td>Tempo</td>
<td>1.9s</td>
<td>2.3s</td>
<td>2.0s</td>
<td>7.1s</td>
<td>5.7s</td>
</tr>
</tbody>
</table>

A.2.2.1 Benchmark de circuitos ISCAS’85

Um estudo sobre a ocorrência de funções RPO sobre circuitos do benchmark ISCAS’85 (IWLS, 2005) foi realizado. A análise foi realizada, a fim de descobrir a frequência de funções RPO, em comparação com funções RO em circuitos mapeados. Foram extraídas funções até 8 entradas dos circuitos através do método K-Cuts (MARTINELLO, 2010). Estas funções foram agrupadas em P-classes, por equivalência
através de permutações de entrada. O termo 'ocorrências' representa o número de funções contadas antes do agrupamento em P-classes.

Tabela A.2.2: Análise de funções até 8 entradas identificadas em circuitos ISCAS’85.

<table>
<thead>
<tr>
<th>Entradas</th>
<th>RO P-Classes</th>
<th>RO Ocorrências</th>
<th>RPO P-Classes</th>
<th>RPO Ocorrências</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>67%</td>
<td>84%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>3</td>
<td>53%</td>
<td>66%</td>
<td>90%</td>
<td>88%</td>
</tr>
<tr>
<td>4</td>
<td>44%</td>
<td>54%</td>
<td>85%</td>
<td>71%</td>
</tr>
<tr>
<td>5</td>
<td>37%</td>
<td>42%</td>
<td>69%</td>
<td>53%</td>
</tr>
<tr>
<td>6</td>
<td>33%</td>
<td>36%</td>
<td>57%</td>
<td>46%</td>
</tr>
<tr>
<td>7</td>
<td>34%</td>
<td>36%</td>
<td>52%</td>
<td>47%</td>
</tr>
<tr>
<td>8</td>
<td>32%</td>
<td>34%</td>
<td>46%</td>
<td>44%</td>
</tr>
</tbody>
</table>

As funções foram extraídas de duas maneiras. A Tabela A.2.2 resume os resultados em relação a todas as funções possíveis de até 8 entradas nos circuitos. Na Tabela A.2.3 foram extraídas as funções dos circuitos que utilizam um algoritmo de cobertura guloso de AIGs (MARTINELLO, 2010).

Da mesma forma que as funções de RO, o número de funções RPO diminui à medida que o número de variáveis aumenta. Este é um resultado esperado, já que quanto maior o número de entradas, maior é a complexidade de implementação da função Booleana.

Tabela A.2.3: Funções selecionadas de circuitos ISCAS’85 utilizando um algoritmo de cobertura gulosa.

<table>
<thead>
<tr>
<th>Entradas</th>
<th>RO P Classes</th>
<th>RO Ocorrências</th>
<th>RPO P Classes</th>
<th>RPO Ocorrências</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>78%</td>
<td>93%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>3</td>
<td>59%</td>
<td>63%</td>
<td>87%</td>
<td>94%</td>
</tr>
<tr>
<td>4</td>
<td>49%</td>
<td>53%</td>
<td>78%</td>
<td>81%</td>
</tr>
<tr>
<td>5</td>
<td>35%</td>
<td>36%</td>
<td>79%</td>
<td>81%</td>
</tr>
<tr>
<td>6</td>
<td>41%</td>
<td>39%</td>
<td>63%</td>
<td>60%</td>
</tr>
<tr>
<td>7</td>
<td>45%</td>
<td>43%</td>
<td>62%</td>
<td>57%</td>
</tr>
<tr>
<td>8</td>
<td>14%</td>
<td>15%</td>
<td>27%</td>
<td>27%</td>
</tr>
</tbody>
</table>

Na Tabela A.2.4, é possível ver o tempo de execução para sintetizar todos os K-cuts (para K = 8). Na Tabela A.2.5, mostra-se o tempo de execução do algoritmo para sintetizar as funções extraídas do circuito coberto.
Tabela A.2.4: Tempo de execução para a síntese de todos os K-Cuts (K=8).

<table>
<thead>
<tr>
<th>Circuit</th>
<th>P Classes</th>
<th>Time (s)</th>
<th>Avg. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1355</td>
<td>680</td>
<td>123.455</td>
<td>0.18</td>
</tr>
<tr>
<td>C17</td>
<td>12</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>C1908</td>
<td>1224</td>
<td>133.162</td>
<td>0.11</td>
</tr>
<tr>
<td>C2670</td>
<td>6345</td>
<td>284.626</td>
<td>0.04</td>
</tr>
<tr>
<td>C3540</td>
<td>9275</td>
<td>123.945</td>
<td>0.01</td>
</tr>
<tr>
<td>C432</td>
<td>844</td>
<td>3.259</td>
<td>0.01</td>
</tr>
<tr>
<td>C499</td>
<td>432</td>
<td>98.187</td>
<td>0.23</td>
</tr>
<tr>
<td>C5315</td>
<td>11350</td>
<td>806.489</td>
<td>0.07</td>
</tr>
<tr>
<td>C6288</td>
<td>142</td>
<td>0.634</td>
<td>0.01</td>
</tr>
<tr>
<td>C7552</td>
<td>17888</td>
<td>8045.137</td>
<td>0.45</td>
</tr>
<tr>
<td>C880</td>
<td>1691</td>
<td>86.301</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Os resultados experimentais sobre circuitos ISCAS'85 demonstraram que as funções RPO são significativamente mais frequentes do que funções RO. Todo o fluxo para fatorar funções RPO foi validado, onde nossa implementação foi capaz de encontrar as soluções exatas para funções de até 8 variáveis binate em um razoável tempo de execução.

Tabela A.2.5: Tempo de execução para a síntese de funções selecionadas através da algoritmos de cobertura gulosa.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>P Classes</th>
<th>Time (s)</th>
<th>Avg. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1355</td>
<td>16</td>
<td>3.70</td>
<td>0.23</td>
</tr>
<tr>
<td>C17</td>
<td>3</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>C1908</td>
<td>44</td>
<td>2.37</td>
<td>0.05</td>
</tr>
<tr>
<td>C2670</td>
<td>68</td>
<td>39.53</td>
<td>0.58</td>
</tr>
<tr>
<td>C3540</td>
<td>129</td>
<td>1.29</td>
<td>0.01</td>
</tr>
<tr>
<td>C432</td>
<td>25</td>
<td>0.15</td>
<td>0.01</td>
</tr>
<tr>
<td>C499</td>
<td>10</td>
<td>0.89</td>
<td>0.09</td>
</tr>
<tr>
<td>C5315</td>
<td>108</td>
<td>58.07</td>
<td>0.54</td>
</tr>
<tr>
<td>C6288</td>
<td>38</td>
<td>0.20</td>
<td>0.01</td>
</tr>
<tr>
<td>C7552</td>
<td>130</td>
<td>8.42</td>
<td>0.06</td>
</tr>
<tr>
<td>C880</td>
<td>36</td>
<td>18.08</td>
<td>0.50</td>
</tr>
</tbody>
</table>
A.3 Conclusões

A principal contribuição deste trabalho foi a definição do conceito de funções read-polarity-once (RPO). Além da definição da classe de funções RPO, várias contribuições relacionadas também foram apresentadas: (1) um algoritmo para fatoração de funções incompletamente especificadas em equações read-once (RO), (2) uma transformação de domínio, que divide variáveis binate em duas variáveis unate independentes e (3) um algoritmo completo para a fatoração exata de funções da classe RPO.

A primeira contribuição foi a proposta de um algoritmo (ISF2RO) que é capaz de fatorar funções Booleanas incompletamente especificadas em formas RO, sempre que possível. O único método conhecido que garante a exatidão do resultado é a abordagem Exact Factor, proposto por (YOSHIDA; Fujita, 2011). No entanto, o método só é viável para funções com até 10 variáveis. O método ISF2RO aqui proposto é capaz de fatorar funções com até 16 variáveis de entrada.

A segunda contribuição foi a proposta de um método para a transformação de domínio que divide variáveis binate em duas variáveis unate independentes, chamado processo de unatization. O processo de unatization é um passo fundamental ao reconhecer funções RPO. No entanto, ele pode ser aplicado em outros métodos de síntese de lógica, como métodos de decomposição funcional.

A terceira contribuição foi o desenvolvimento de um algoritmo completo para a fatoração exata de funções da classe RPO, chamado ISF2RPO. Tal algoritmo foi implementado e comparado com algoritmos de fatoração existentes. O algoritmo proposto garante formas fatoradas mínimas para a classe de funções RPO. Os resultados mostram que, para o conjunto de 612.125 funções da classe NPN com até 5 entradas, 1.462 funções RPO foram identificadas, enquanto que para a classe RO apenas 21 funções foram identificadas. Além disso, os resultados levando em conta circuitos ISCAS'85 mostraram que as funções RPO são muito mais frequentes em relação às RO.

Vale a pena ressaltar que ambos os algoritmos foram implementados em uma ferramenta, chamada Switchcraft (CALLEGARO et al, 2010). O ambiente Switchcraft fornece um conjunto de ferramentas para a geração de redes de chaves e de portas lógicas. A ferramenta foi utilizada e aplicada em aula na Universidade Federal do Rio Grande do Sul (UFRGS) no Brasil, em cursos de graduação e pós-graduação.