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

AUGUSTO NEUTZLING

Threshold Logic Technology Mapping for Emerging Nanotechnologies

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

Advisor: Prof. Dr. Renato Ribas

Porto Alegre
November 2017

CIP — CATALOGING-IN-PUBLICATION

Neutzling, Augusto

Threshold Logic Technology Mapping for Emerging Nanotechnologies / Augusto Neutzling. – Porto Alegre: PPGC da UFRGS, 2017.

85 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: Renato Ribas.

1. Logic Synthesis. 2. Nanotechnologies. 3. Majority logic. 4. Threshold logic. 5. Technology mapping. 6. Digital circuit. I. Ribas, Renato. II. Título.

ABSTRACT

Threshold logic is a powerful alternative paradigm for realizing Boolean functions in digital circuit design. A threshold logic function (TLF) can be roughly defined as a Boolean function in which the output is evaluated in terms of input weights and a threshold value. Although the subject has been investigated since the 1960's, the lack of effective hardware implementation for threshold functions led to a loss of interest in developing a threshold logic design flow. However, for some emerging technologies, such as memristors, spintronic, quantum cellular automata (QCA) and resonant tunneling devices (RTD), such a logic design strategy seems to be more appropriate than the traditional switch-based CMOS circuitry. Thus, research and development of synthesis and verification methods applicable to large, multi-level threshold circuits are desired. Existing state-of-the-art threshold logic synthesis tools rely on locally resynthesizing each single-output node out of circuits initially mapped disregarding thresholdness. This work presents the first effective technology mapping approach for threshold logic gates (TLGs), which is based on identifying threshold logic functions during the mapping. This enables to explore the entire circuit-level search space, seeking a threshold logic covering. As a consequence, we improve both area and performance results, as well as the synthesis scalability. A second contribution introduced in this thesis improves the quality of results by efficiently exploring redundant cuts. The technology mapper, we propose herein, is also able to target different threshold-based area estimations: the total summation of input weights and threshold values; the total summation of gate inputs; and the total number of TLGs. Finally, we propose a TLF-based approach to perform logic synthesis for majority-gate-based emerging nanotechnologies.

Keywords: Logic Synthesis. nanotechnologies. majority logic. threshold logic. technology mapping. digital circuit.

Síntese Lógica para nanotecnologias emergentes

RESUMO

Lógica de Limiar (*Threshold Logic*) é um promissor paradigma alternativo para implementar funções Booleanas is projetos de circuitos digitais. Uma função limiar pode ser definida como uma função Booleana onde a saída é avaliada em termos dos pesos das entradas e um valor de threshold. Embora esse assunto tenha sido investigado desde a década de 1960, a lacuna por implementações em hardware eficientes para funções threshold resultaram em um menor interesse no desenvolvimento de um fluxo de projeto baseado em *threshold logic*. No entanto, para algumas tecnologias emergentes como memristors, spintronic e diodos de tunelamento ressonantes (RTD), essa estratégia de projeto se mostra mais apropriada que os circuitos CMOS tradicionais baseados em chaves lógicas. Portanto, a pesquisa e o desenvolvimentos de métodos de síntese e verificação aplicáveis a circuitos *threshold* multi-níveis são necessárias. As ferramentas estado-da-arte para a síntese de circuitos *threshold* realizam um mapeamento tecnologico genérico, sem considerar informações de propriedades *threshold*, e depois realizam uma resíntese para cada nodo do circuito mapeado. Este trabalho apresenta a primeira abordagem efetiva de mapeamento tecnológico para portas lógicas *threshold* (TLGs), baseada em identificar funções *threshold* durante o mapeamento. Essa abordagem habilita a exploração do espaço de busca em todo o circuito, procurando por uma cobertutra *threshold logic*. Como consequência, os resultados em termos de área e desempenho são melhorados, assim como a escalabilidade do circuito. Uma segunda contribuição introduzida nesse trabalho é melhora da qualidade dos resultados explorando cortes redundantes de uma maneira mais eficiente. Finalmente, o mapeador tecnológico proposto também é capaz de otimizar diferente estimativas de área dos TLGs: o somatório total de pesos e valor de *threshold*; o somatório total de entradas; e o número total de TLGs.

Palavras-chave:

.

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-Inverter Graph
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
CAD	Computer Aided Design
CMOS	Complementary Metal-Oxide-Semiconductor
DAG	Directed Acyclic Graph
EDA	Electrical Design Automation
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
ILP	Integer Linear Programming
MAJ	Majority gate
NCL	Null Convention Logic
QCA	Quantum Cellular Automata
RTD	Resonant Tunneling Diode
RTL	Register-Transfer Level
SET	Single Electron Transistor
SOP	Sum-Of-Products
TPL	Tunneling Phase Logic
VLSI	Very-Large Scale Integration
ZBDD	Zero-Suppressed Binary Decision Diagram

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The MOS transistor begins to reach its physical limits of construction, with dimensions very close to the atomic dimensions of the Silicon (Si), which is the basis for the realization of this device. The estimated limit for the length of the transistor channel is about 5 nm. It is also estimated that, in the next 5-10 years, the MOS transistor will be replaced by new devices that allow continue in the evolution of integrated circuits according to the prediction of Moore's Law (ITRS, 2015).

An alternative for the technological evolution is not dedicating the main development effort in reducing the MOS transistor, but look for new alternatives in different technologies. Among new nanoscale devices, candidates to replace the MOS transistor, we can mention the memristor (memory resistor)(GAO; ALIBART; STRUKOV, 2013; FAN; SHARAD; ROY, 2014), spintronic (NUKALA; KULKARNI; VRUDHULA, 2014), quantum cellular automata (QCA) (CAMPOS et al., 2016a), tunneling phase logic (TPL), single electron tunneling (SET), resonant tunneling diode (RTD), graphene transistor and carbon nanotubes. Besides, there are other technologies based on the same MOS transistor devices, such as SiGe, SiC, FinFET, ZnO, etc (KAHNG, 2013).

If the candidate who will replace the MOS transistor in the future has a similar behavior, based on operation of ideal logic switches type P and type N, then the current knowledge and the existing design and optimization CAD tools can be applied directly on such a new technology. And, in this case, scientific and industrial effort for technological advancement can focus only on an appropriate and reproducible physical construction of the new device.

However, for some of the mentioned new technologies, it is already possible to notice new challenges in implementing logic functions and digital circuits. For instance, in the case of QCA technology, the logic gate which requires less physical area for its construction is the 3-input majority gate (LENT et al., 1993). Something similar happens with memristor and spintronic technologies, where the most optimized core circuit of a logic function is a threshold logic gate (MAAN; JAYADEVI; JAMES, 2017).

The fact is that research on new methods and paradigms of logic synthesis, which are more appropriate to future candidates to replace MOS technology, should not be delayed until the completion of this substitution. It would generate a delay in the development of integrated circuits due to the waiting for the development of knowledge and circuit design tools for a specific technology.

1.1 Motivational case: differences between threshold and CMOS logic

Threshold logic is a powerful alternative paradigm for building Boolean functions in digital designs. A threshold logic function (TLF) can be roughly defined as a Boolean function in which the output is evaluated in terms of input weights and a threshold value. A TLF is completely represented by a compact vector $[w_1,w_2,\ldots,w_n;T]$, where $w_1,w_2,\ldots,w_n$ are the input weights and T is the function threshold value. For instance, the representation of the given functions $f = x_1x_2x_3$ and $g = x_1 + x_2 + x_3$ are $[1,1,1;3]$ and $[1,1,1;1]$, respectively. The electronic structure which implements a TLF is called a threshold logic gate (TLG).

The main challenges when designing threshold logic based circuits are the differences to that based on traditional static CMOS logic. For instance, Boolean functions which are trivially implemented in a CMOS logic gate, as the function $f = x_1x_2 + x_3x_4 + x_5x_6$, requires a network of TLGs. In the other hand, Boolean functions considered complex to be implemented in CMOS can be easily implemented in a single TLG. In the following, we discuss an example for illustrating this challenge. Notice that the objective is not to explain how the gate works, but demonstrate the differences between threshold logic and CMOS based circuits. Figure 1.1 presents an RTD-based TLG, where each device area corresponds to an input weight or to the threshold value.

T=5	F_1=*abcde*
T=4	F_2=*abcd+abce+abde+acde+bcde*
T=3	F_3=*abc+abd+abe+acd+ace+ade+bcd+bce+bde+cde*
T=2	F_4=*ab+ac+ad+ae+bc+bd+be+cd+ce+de*
T=1	F_5=*a+b+c+d+e*

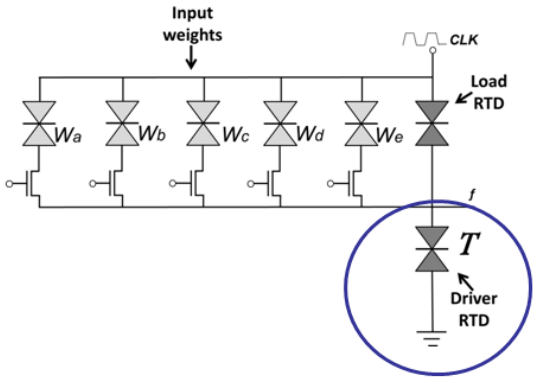


Figure 1.1: Five different Boolean functions implemented in RTD-based threshold logic gate by just when varying the threshold value (*i.e.* the drive device sizing.

Suppose all the input weights w_a, w_b, w_c, w_d and w_e be equal to 1 and the threshold value T equal to 5, *i.e.*, $F_1 = [1, 1, 1, 1, 1; 5]$. This corresponds to the Boolean function $F_1 = abcde$, the 5-input AND (AVEDILLO; J.M., 2004). By keeping the input weights and by changing only the threshold value, one can implement different functions. For instance, by decreasing the threshold value to 4, the gate implements the function $F_2 = abcd + abce + abde + acde + bcde$, and by decreasing the threshold value to 3, the gate implements $F_2 = abc + abd + abe + acd + ace + ade + bcd + bce + bde + cde$. Figure 1.1 shows different functions implemented by keeping all input weights equal to 1 and just decreasing the threshold value.

Whereas, in threshold logic, the design cost for implementing the more complex functions is practically the same to implement the 5-input AND/OR ones, in CMOS the implementation of these functions are significantly different. Figure 1.2 shows the implementation of the same 5-input Boolean functions in static CMOS logic. For instance, the number of transistors used to implement F_1, F_2 and F_3 is 10, 21 and 46, respectively. The circuit complexity varies significantly. Such differences should be taken into account when developing algorithms to design a TLG-based circuit.

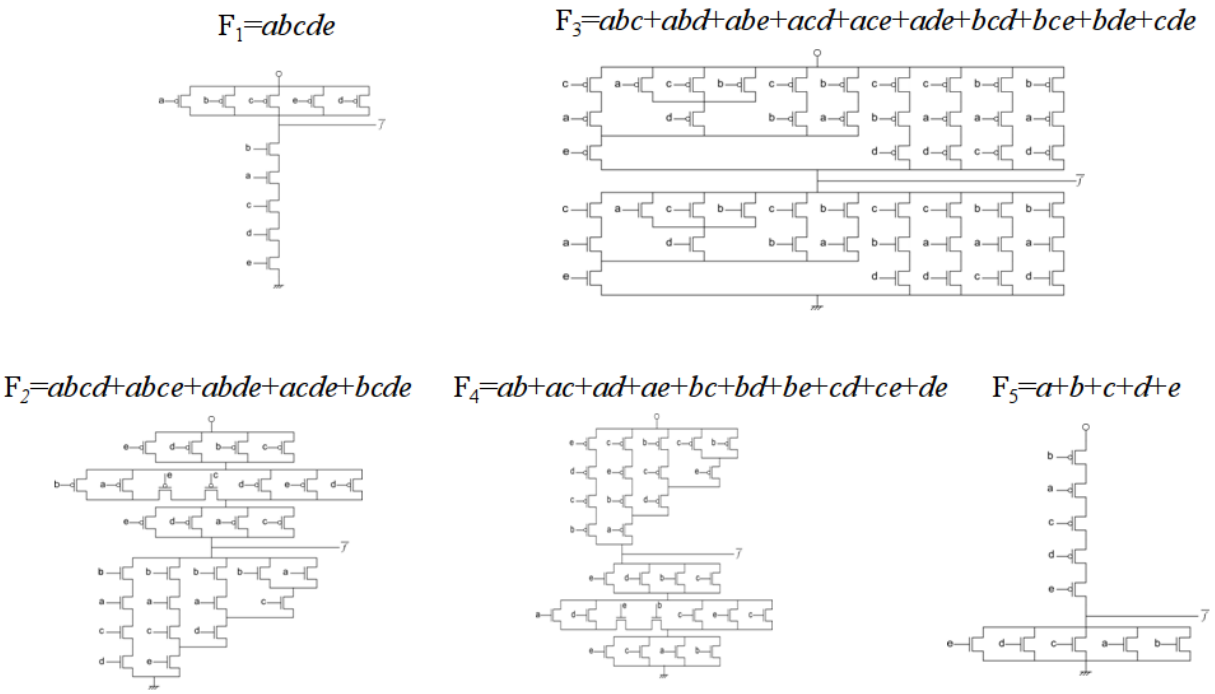


Figure 1.2: Five different Boolean functions implemented in traditional static CMOS gate.

1.2 Overview of the related works

Although the subject has been studied since the 1960's, the lack of effective hardware implementation for threshold functions led to a loss of interest in developing a threshold logic design flow (MUROGA, 1971). However, for some of the emerging technologies, such as memristors (GAO; ALIBART; STRUKOV, 2013; FAN; SHARAD; ROY, 2014), spintronic (NUKALA; KULKARNI; VRUDHULA, 2014) and resonant tunneling devices (RTD) (AVEDILLO; QUINTANA, 2004), such a logic design strategy seems to be more appropriate than the traditional switch-based CMOS circuitry. Thus, research and development of synthesis and verification methods applicable to large, multi-level threshold circuits are desired.

Algorithms addressing threshold logic synthesis have been presented in recent years (ZHANG et al., 2005; SUBIRATS; JEREZ; FRANCO, 2008; GOWDA et al., 2011; PALANISWAMY; TRAGOUDAS, 2014; NEUTZLING et al., 2014; LIN et al., 2014; CHEN; WANG; CHANG, 2016). The main drawback of these methods is that they do not consider threshold logic while generating an initial covering in terms of 6-input single-output nodes. Once the circuit is already covered, the previous approaches locally perform a threshold network synthesis for each single-output node, aiming to cover the circuit using only threshold logic gates (TLGs). Besides, their area estimation relies only on the number of TLGs in the final circuit. This estimation may be inaccurate, since some TLG implementations fit better with area estimations related with the sum of input weights and threshold values, or even the total number of inputs.

1.3 Thesis proposal

The objective of this thesis is to propose an effective technology mapping for emerging nanotechnologies that are based on threshold and majority logic functions. In this sense, effective means to explore the thresholdness proprieties of the target circuit earlier in the flow. As a consequence, the method can explore solutions that are impossible to be reached for the previous works. In the following, a case of study of this improvement is presented. Fig. 1.3 presents a circuit with 3 inputs and 2 outputs is represented through an And-Inverter Graph logic structure called AIG. An output implements a 2-input Exclusive-OR (XOR) between the inputs a and b, and the other output implements a 2:1 multiplexer (MUX), being the input b the selector.

Since the first step of previous approach is to perform a general technology mapping, *i.e.*, without considering thresholdness information, such approaches obtain a network containing one LUT for each output. This solution, presented in Fig.1.3, is optimal in terms of LUTs (2 LUTs) and in terms of logic depth (1 level). However, as demonstrated in Fig 1.4, this solution results in 4 TLGs, since each LUT implementation requires 2 TLGs.

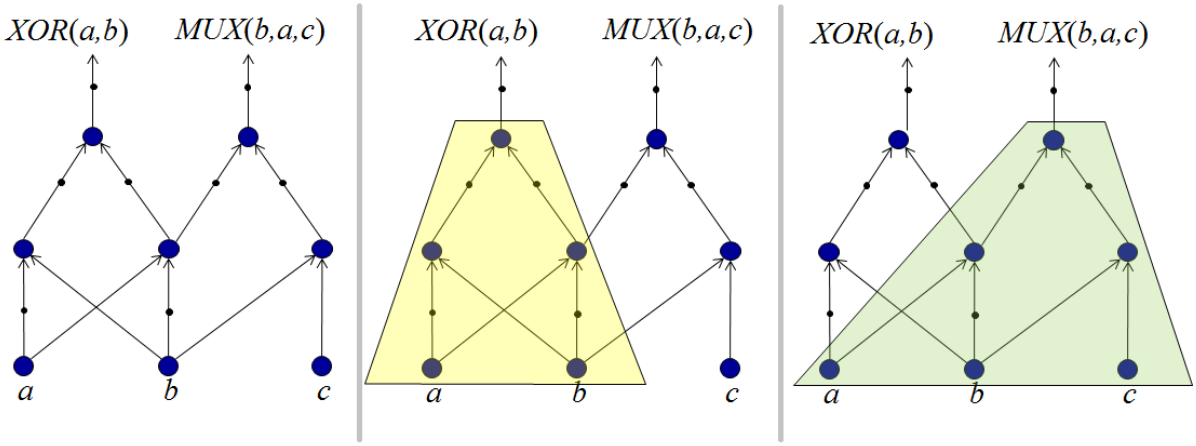


Figure 1.3: The best general technology mapping for the given circuit comprises only one LUT for each output.

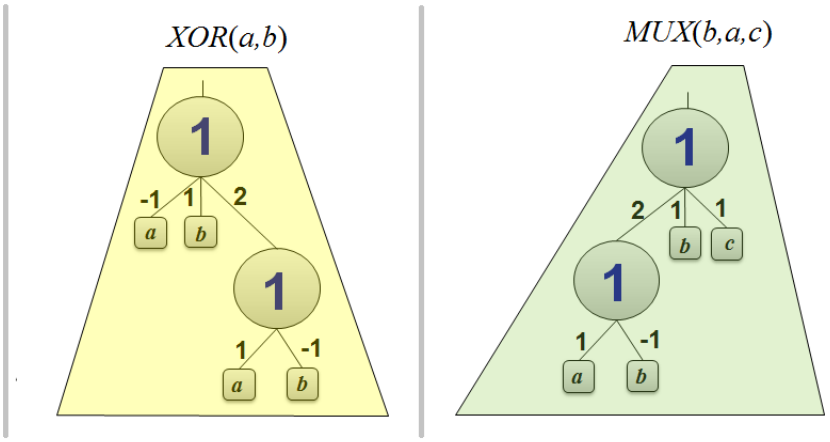


Figure 1.4: The synthesis is performed independently for each LUT. The best solution obtained by traditional approaches comprises 4 TLGs.

In the other hand, in our approach, during the technology mapping we already know which "parts" of the AIG can be implemented in a single TLG or not. As a consequence, the solution of the proposed approach contains 3 LUTs and 2 levels of depth .

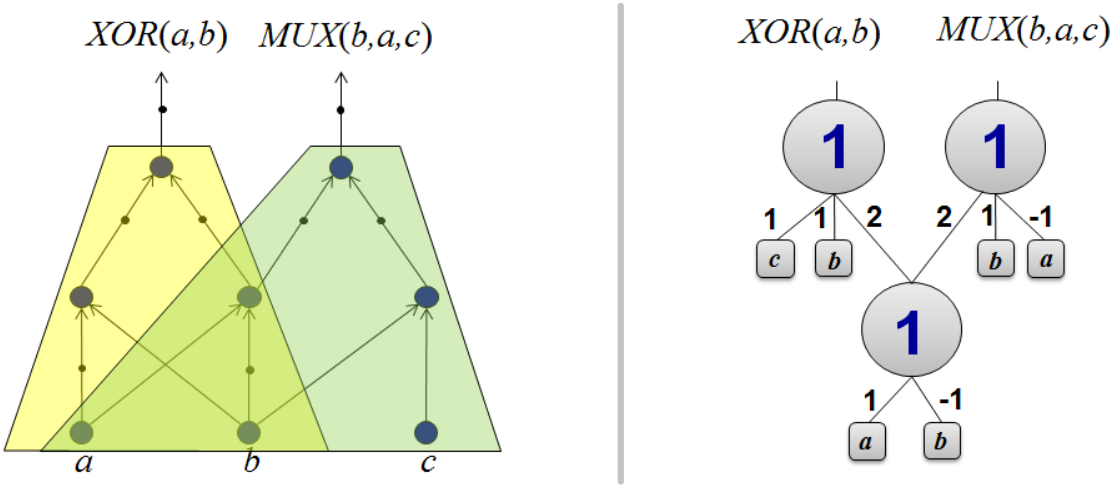


Figure 1.5: The proposed flow knows the thresholdness information during the technology mapping. The solution comprises 3 TLGs.

In this case, each LUT function is a TLF, *i.e.*, corresponds directly to a TLG in the final netlist, as shown in Fig. 1.5.

Besides to enable improvements, as illustrated in Fig. 1.5, the threshold technology mapping enables different improvements on logic synthesis for emerging technologies. These improvements, pointed out in the following, are the specific objectives of the thesis.

1.3.1 Specific objectives

The proposed threshold logic technology mapping enables several opportunities of both improvements and new approaches based on threshold logic synthesis.

- We present a novel threshold logic synthesis flow combining area and delay optimization. In particular, the proposed strategy relies on identifying threshold logic functions earlier in the mapping flow. This simple (yet effective) change gives us a fast and clean synthesis flow and a better quality-of-results (QoR) based on exploring the threshold logic information during the technology mapping.

- The TLF-based technology mapping is based on priority cuts, where the main task is the cut sorting. We propose different approaches for cut sorting based on the thresholdness and unateness information.

- We are also exploring QoR improvments for threshold synthesis by cleverly enumerating redundant cuts. We show that, even if a given irredudant cut i does not

represent a threshold function, there may exist some threshold cuts redundant with i which can then be used by the threshold logic mapper to reduce the circuit area.

- The technology mapper, we propose herein, is also able to target different threshold-based area estimations: the total sum of input weights and threshold values, the total sum of gate inputs, or the total number of TLGs. This is particularly interesting when the synthesis targets specific threshold-oriented emerging technologies, in which the TLG area estimation varies from one technology to another.

- Finally, we propose a majority gate synthesis approach, which is based on the TLF technology mapping and explores the relationship between threshold and majority functions.

1.4 Thesis Structure

The rest of the text is organized as follows.

- In Section 2, some fundamentals on Boolean networks, AIGs and structural cuts are reviewed for a better understanding of the proposed approach. We define threshold logic functions and threshold logic gates, besides discussing different TLG area estimations used in the proposed mapper.

- Section 3 presents the related works. This section is essential to identify how the previous approaches perform threshold logic synthesis and understand the contributions of the proposed thesis.

- The proposed threshold synthesis flow is presented in Section 4. We explain the proposed technology mapping, based on the LUT based mapping. The two main tasks are detailed: the cut enumeration and cut covering. We present several orthogonal improvements to TLG-oriented technology mapping, such as (i) threshold logic priority cuts; (ii) redundant threshold cuts; (iii) optimizing different TLG area estimations.

- Section 5 provides the results, demonstrating the potential of the proposed method when comparing to the state-of-the-art approaches. We present results which support each of the improvements discussed on Section 4. Moreover, we present new results over huge benchmarks which can be adopted in future works evaluation.

- In Section 6, we present two additional contribution:

  (i) Some new technologies are suitable for a specific class of threshold logic functions, the majority functions (*a.k.a.* as voters). We propose a logic synthesis for majority functions based on the threshold logic synthesis flow;

  (ii) A crucial task for any threshold logic synthesis flow is the threshold logic identification. We analyze the trade-offs between ILP and heuristic approaches, proposing a fast and efficient approach, which is an enabler for the threshold logic technology mapping proposed in this thesis.

- In Section 7, we present the final considerations, pointing out the main contributions and discussing open questions to be addressed in future works .

2 PRELIMINARIES

In this section, some fundamentals on Boolean networks, AIGs and structural cuts are reviewed. We define threshold logic functions and threshold logic gates. The background for the different area estimations considered in this work is presented. We also review LUT-based technology mapping for FPGAs used in the proposed mapper.

2.1 General Terms and Definitions

2.1.1 Boolean Function

A Boolean function f defined over the variable set $X = \{x_1,\ldots,x_n\}$ is a function defined as $f(X)$: $B^n \rightarrow B$, where $B = \{0,1\}$ and $n = |X|$, i.e., n is the number of variables in X. In this work, AND, OR and NOT operations are denoted by '.','$\bigvee$' and '!', respectively.

2.1.2 Cubes and Literals

A literal is a variable (x_i) or its complement ($!x_i$), whereas a cube is a product of literals that represents a Boolean sub-space. The cube size of a cube with l literals in a Boolean space Bn is given by $2^{(n-l)}$. Consequently, the size of a cube is inversely proportional to the number of literals in this cube.

2.1.3 Sum-of-Products

Furthermore, an expression is called sum-of-products (SOP) when this expression corresponds to product terms (AND) joined by a sum (OR) operation. In particular, an irredundant sum-of-products (ISOP) is a SOP in which neither a literal nor a cube can be removed without changing the function behavior.

2.1.4 Unateness

A completely specified function is positive unate in variable x, iff $f(x_i = 1) \supset f(x_i = 0)$, where $f(x_i = 1)$ is the positive cofactor and $f(x_i = 0)$ is the negative cofactor of f with respect to variable x_i. Given a Boolean function, the unateness checking is based on the positive and negative cofactors generation. Given a function f represented by an ISOP form, this function is unate if, and only if, either direct (positive polarity) or complemented (negative polarity) literals, but not both, appear to each variable. Notice that a unate function has a unique ISOP representation (BRAYTON et al., 1982).

2.1.5 Boolean Network

A Boolean network is a directed acyclic graph (DAG) where nodes correspond to logic gates and directed edges represent the wires connecting the gates. It is assumed that each node has a unique ID (integer number).

A *fanin* (*fanout*) cone of node n is a subset of all nodes of the network reachable through the *fanin* (*fanout*) edges from the given node.

A node n has zero or more *fanins* (nodes driving n) and zero or more *fanouts* (nodes driven by n). The primary inputs (PIs) are nodes without *fanins*, whereas the primary outputs (POs) are a subset of nodes from the network connecting it to the environment.

2.1.6 AIG

AND-inverter graph (AIG) is a DAG where each node has either zero incoming edges (in the case of a PI) or two incoming edges (in the case of an AND node). Each edge can be complemented or not. Some nodes are marked as POs.

2.1.7 Structural Cuts

A *cut* C of a node n is a set of nodes of the network, called *leaves* of the cut, such that every path between a PI and n contains a node in C. A cut of n is irredundant if no subset on it is a cut. A K-feasible cut contains K or fewer nodes. Fig. 2.1 present some examples of K-cuts when limiting $Kto3$.

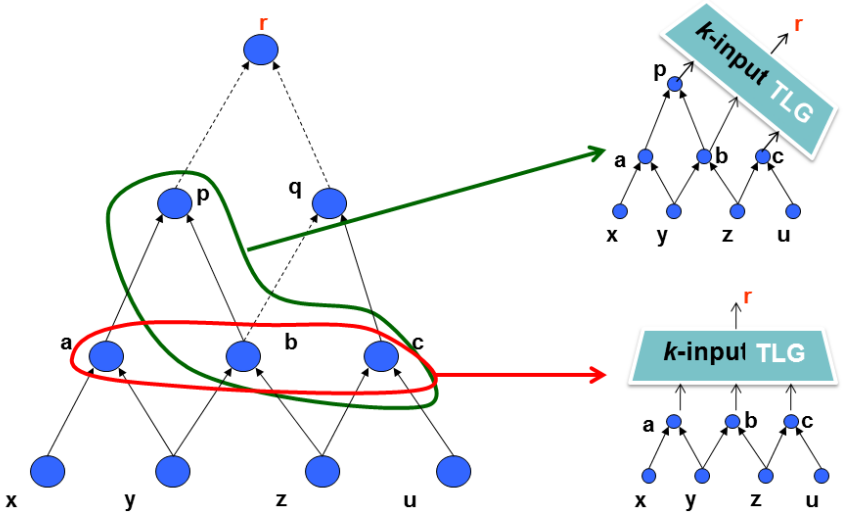


Figure 2.1: Examples of structural cuts: sets of nodes C of the network such that every path between a PI and n contains a node in C.

Node n is called the root of cut C. The cut size is the number of its leaves. A trivial cut is the node itself. A local function of an AIG node n, denoted by $f_n(x)$, is a Boolean function of the logic cone rooted in n and expressed in terms of the leaves x of a cut of n.

Cut enumeration is a technique used by a cut-based technology mapper to perform cut computation using dynamic programming, starting from PIs and ending at POs (MISHCHENKO; CHATTERJEE; BRAYTON, 2007; PAN; LIN, 1998).

2.1.8 NPN Classes

By considering a set of all functions with up to n variables, these functions can be grouped into classes. Boolean functions can be grouped taking into account the negation (N), and/or the permutation (P) of variables, and/or the negation of function value (HINSBERGER; KOLLA, 1998). For instance, NP-class corresponds to the set of distinct functions obtained by negating and/or permuting the input variables.

2.2 Threshold Logic Terms and Definitions

2.2.1 Threshold Logic Function

A threshold logic function is a Boolean function satisfying the following condition. Each input has a specific weight and the gate has a threshold value. If the weight sum of active inputs (inputs equal to 1) is equal or greater than the threshold value, the function evaluates to 1. Otherwise, the function evaluates to 0. This can be expressed as follows (MUROGA, 1971):

$$f = \begin{cases} 1, & \textit{if } \sum_{i=1}^{n} x_i w_i \geq T; \\ 0, & \textit{otherwise}, \end{cases} \tag{2.1}$$

where x_i represents each Boolean input value $\{0, 1\}$, w_i is the weight of each input, and T is the function threshold value.

A TLF is completely represented by a compact vector $[w_1,w_2,\ldots,w_n;T]$, where $w_1,w_2,\ldots,w_n$ are the input weights and T is the function threshold value. For instance, the corresponding TLG of the given functions $f = x_1x_2x_3$ and $g = x_1 + x_2 + x_3$ are $[1, 1, 1; 3]$ and $[1, 1, 1; 1]$, respectively. A TLF can also be called a 'linearly separable' function.

2.2.2 Threshold Logic Identification

Although some complex functions are TLFs, there exist some simple functions which are not TLFs. For instance, the function $h = x_1x_2 + x_3x_4$ cannot be represented in terms of input weights and threshold value. The threshold logic identification process verifies if a Boolean function is TLF (or not) and compute the input weights and gate threshold value. In this work, we adopt the identification process presented in (NEUTZLING et al., 2013), instead of the integer linear programming based algorithms applied in previous works (ZHANG et al., 2005; SUBIRATS; JEREZ; FRANCO, 2008, 2008; GOWDA et al., 2011; PALANISWAMY; TRAGOUDAS, 2014). This is mainly due to the fast runtime and the good quality of results.

2.2.3 Threshold Logic Properties

All threshold logic functions are unate, *i.e.*, if a given function is not unate then it is not TLF. Unate functions can be non-threshold functions. Considering threshold functions, a negative unate variable can be changed to a positive one by just inverting the weight signal, and this amount is then subtracted from the threshold value.

In identification methods, negative variables are usually treated as positive ones, and this information is stored. After computing the threshold function parameters, the input weights are adjusted.

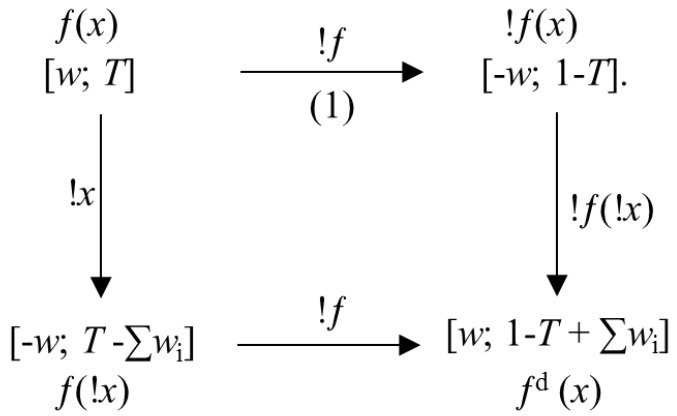


Figure 2.2: Elementary properties of threshold logic functions (MUROGA, 1971).

For instance, in the given function $f_{original} = (!a \bigvee (b.c))$, the variable a is negative unate. The method considers the function $f_{positive} = (a \bigvee (b.c))$, where a is positive unate. The identified variable weights for $f_positive$ are 2, 1 and 1. Afterwards, the signal of the negative variables and the function threshold value are adjusted based on the properties illustrated in Fig. 1. The variable weights for $f_original$ are -2, 1 and 1, respectively.

2.2.4 Threshold Logic Gates

More than a gate that realizes a threshold function, a threshold logic gate is a single primitive, or a non-decomposable circuit, which physically embodies the comparison expressed in Equation (2.1). Notice that this excludes implementations that realize the threshold functions using CMOS-like, AND/OR-based primitives.

TLGs can implement complex functions. For instance, the TLG [4,3,3,1,1;7] implements $f = x_1x_2 + x_1x_3 + x_2x_3x_4 + x_2x_3x_5$. Using larger threshold functions has the

potential benefit of reducing the total number of gates needed to implement digital circuits.

Several implementations of TLGs have been proposed for both CMOS and new nanometric technologies. A survey with more than 50 TLG implementations was presented by BEIU; QUINTANA; AVEDILLO, in (BEIU; QUINTANA; AVEDILLO, 2003). More recently, implementations based on memristors (GAO; ALIBART; STRUKOV, 2013), spintronic devices (FAN; SHARAD; ROY, 2014; NUKALA; KULKARNI; VRUDHULA, 2014), SET (LAGEWEG; COTOFANA; VASSILIADIS, 2001; CHEN; MAO, 2008), QCA (NAVI et al., 2010), and RTDs (ZHANG et al., 2005; PACHA et al., 2000) have also been proposed.

2.2.5 Threshold Logic Network

Threshold logic network (TLN) is a netlist of TLFs and its interconnections. A TLN can be implemented through TLGs, since each TLF can be directly implemented through a single TLG. The area of TLN corresponds to the sum of the TLG areas.

2.2.6 TLG Area Estimation

The area of a TLG depends directly upon the technology used to implement such a gate. However, since no technology currently available is mature enough to implement TLGs in large scale, or in a standard cell library, synthesis tools usually consider that each TLG has the same area. As a consequence, the total circuit area is commonly calculated through the overall number of TLGs in the mapped circuit.

Although it is hard to define a general TLG area estimation, some of them are more suitable for specific technologies. For instance, when implementing a TLG through RTDs, each input weight and the threshold value define the diode area. Therefore, the gate area is directly related to the sum of the weight inputs and the threshold, as defined in the following equation:

$$A_{\mathrm{TLG}} = T + A_u(\sum_{i=1}^{k} w_i), \tag{2.2}$$

where k is the number of inputs of the gate, w_i is the weight of input i, A_u is the unit area of an RTD with $w = 1$, and T is the threshold of the gate (ZHANG et al., 2005).

For other technologies, such as memristors (GAO; ALIBART; STRUKOV, 2013;

FAN; SHARAD; ROY, 2014) or spintronic-based devices (NUKALA; KULKARNI; VRUDHULA, 2014), the weight is set by applying a voltage for some time and, thus, do not impact the device area. In these cases, each input requires one device (with the same area) and, as a consequence, the more suitable gate area estimation is the number of its inputs.

3 RELATED WORKS ON THRESHOLD LOGIC SYNTHESIS

The studies on threshold logic were intense the 1960s (HU, 1965; DERTOUZOS, 1965; LEWIS; COATES, 1967; SHENG, 1969; MUROGA, 1971). The lack of efficient circuit implementations, competitive with static NMOS and later CMOS logic, resulted in less interest in further development of threshold logic synthesis methods. Recently, the interest in threshold logic has been renewed together with emerging nanotechnologies proposed as alternatives to CMOS, such as resonant tunneling diodes (AVEDILLO; QUINTANA, 2004), quantum cellular automata (CAMPOS et al., 2016a), single electron transistors and memristive devices (GAO; ALIBART; STRUKOV, 2013), because can implement threshold functions very compactly and efficiently.

Table 3.1 presents a summary of the related works that are discussed in this section. For each work we present the publication year and vehicle as well as the author's name. The works are joined according to their strategies in three groups: threshold logic synthesis, threshold logic rewriting and hybrid (CMOS and Threshold logic). The table presents also a brief description of the works in these groups.

Table 3.1: Related works grouped according their strategies.

Threshold Logic Synthesis			Threshold Logic Rewriting			Hybrid synthesis		
2004	Avedillo	Euromicro	2014	Lin	DATE	2016	Kulkarni	TVLSI
2005	Zhang	TCAD	2014	Palaniswamy	JETC			
2008	Subirats	TCAS	2016	Lee	ICCAD			
2011	Gowda	TCAD						
2014	Neutzling	ISCAS						
Description: Perform a generic technology mapping and for each non-TLF Boolean function in the resulting network find a TLG network.			**Description:** Starts from a TLG network. Requires another method to perform threshold logic synthesis.			**Description:** Starts from a standard cell netlist and replace standard flip-flops to threshold logic flip-flops.		

Previous methods on threshold logic synthesis start by generating an initial covering in terms of 6-input nodes. Once the circuit is already covered, they locally perform a threshold network synthesis for each single-output node, aiming to cover the circuit using only threshold logic gates (TLGs).

Te first threshold logic synthesis approach in the recent years was proposed by AVEDILLO; QUINTANA in (AVEDILLO; QUINTANA, 2004). This work uses the SIS logic synthesis tool for "partition" the circuit and integer linear programming (ILP) is used to perform threshold logic identification. For each Boolean function, a search procedure

is used to synthesize feedforward threshold circuits with one threshold gate at each level. This procedure is based on SAT formulation. This method is suitable only for small fanin and for small circuits, taking minutes to synthesize them. Moreover, the number of levels is equal to the number of gates, a naive strategy in terms of logic depth. Fig. 3.1 demonstrates the structure of the threshold networks generated by such method.

ZHANG et al., in (ZHANG et al., 2005), propose the recursive partition of non-threshold functions to merge the nodes respecting fanin restrictions. The input to this methodology is an algebraically-factored multi-output combinational Boolean network, generated by SIS tool. AN ILP approach is applied to perform the threshold logic identification.

Two methods are used for synthesis, as shown in Fig. 3.2. In Method 1, the synthesis algorithm begins by processing each primary output of the Boolean network. At first, the node representing a primary output is collapsed. If the node represents a binate function, it is split into multiple nodes which are then processed recursively. If the unate node is a threshold function, it is saved in the threshold network and the fanins of the node are processed recursively. Otherwise, the unate node is first split into two nodes. If neither of the split nodes is a threshold function, the original node is split into multiple nodes which are then processed recursively. The synthesis algorithm by Method 1 terminates when all the nodes in network are mapped into threshold nodes. In Method 2, one-to-one mapping is firstly performed on the network. After that, starting from each primary output, we obtain the subnetworks of AND and OR gates only satisfying the fanin restriction. The synthesis is then performed on each subnetwork by Method 1.

Zhang's approach is very important because it is the first open source available method for threshold logic synthesis. The code was developed inside the SIS tool and

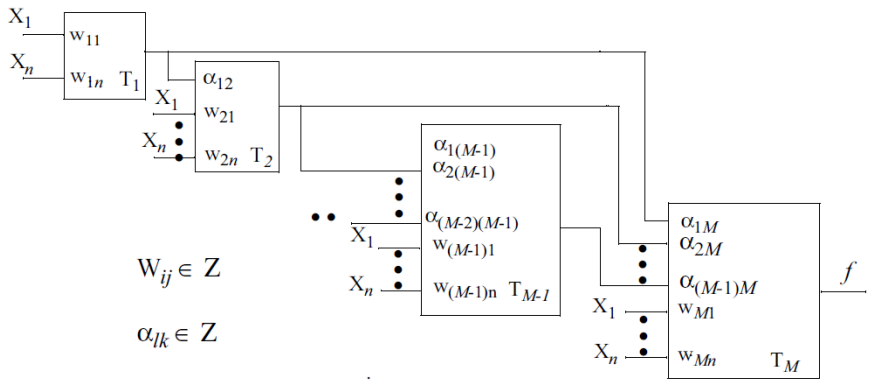


Figure 3.1: Threshold logic network structure from Avedillo's method, with a single TLG per level (AVEDILLO; QUINTANA, 2004).

is public available. For this reason, several works in threshold logic synthesis adopt this work as reference in order to compare their results. The main drawback is that the quality of results is very sensitive to the initial Boolean network description.

Moreover, a method based on truth table descriptions was presented by SUBIRATS; JEREZ; FRANCO, in (SUBIRATS; JEREZ; FRANCO, 2008). This method also starts from a Boolean network generated by SIS tool and applies ILP to perform threshold logic identification. Subirats' algorithm computes a variable ordering using information about the on-set and the off-set, and performs Shannon decomposition in order to find threshold logic functions.

For each Boolean function in the Boolean network, the algorithm is performed. According to its Hamming distance, it is decided whether to use an OR or AND representation of the function. The choice of representation affects the algorithm in two ways: firstly, the output function of the final architecture will be an OR (AND) if more than half of the outputs of the target function are 1's (0's). Secondly, it affects the function splitting procedure, as 0's and 1's are used for filling new undefined instances created after the splitting process is applied. For the case of an input target function with half of the bits equals to 0, any of the two choices can be used.

After the representation to be used is selected, the whole decomposition procedure continues by adding the target function to the work-set, a reservoir that contains all functions that will be analyzed. At the beginning, the work-set contains only the target function but later on, it will contain the newly created functions, after the splitting procedure is applied iteratively. The algorithm continues by picking a function from the work-set and applying a variable simplification procedure in order to eliminate irrelevant

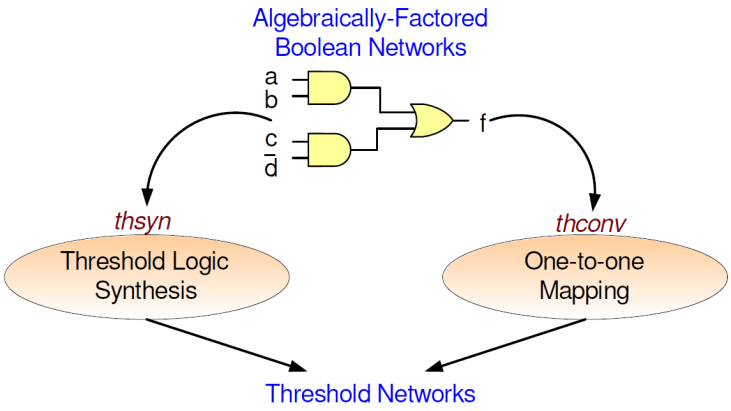


Figure 3.2: Overview of the threshold logic synthesis flow proposed by Zhang *et. al* in (ZHANG et al., 2005).

variables which have no impact at the output of the function. If the function is not TLF the functions are added to the work-set. Otherwise, if the function is TLF, it is added to the solution set. The whole procedure is repeated until there is no more function to be analyzed in the work-set. The set of found threshold functions are then all combined using the OR and AND functions selected at the beginning and the final output will compute the desired function.

Subirat's approach improves the Zhang's results in terms of number of gates. However, this approach produces two-level threshold networks without fanin restriction, which is more suitable for neural networks than for digital designs.

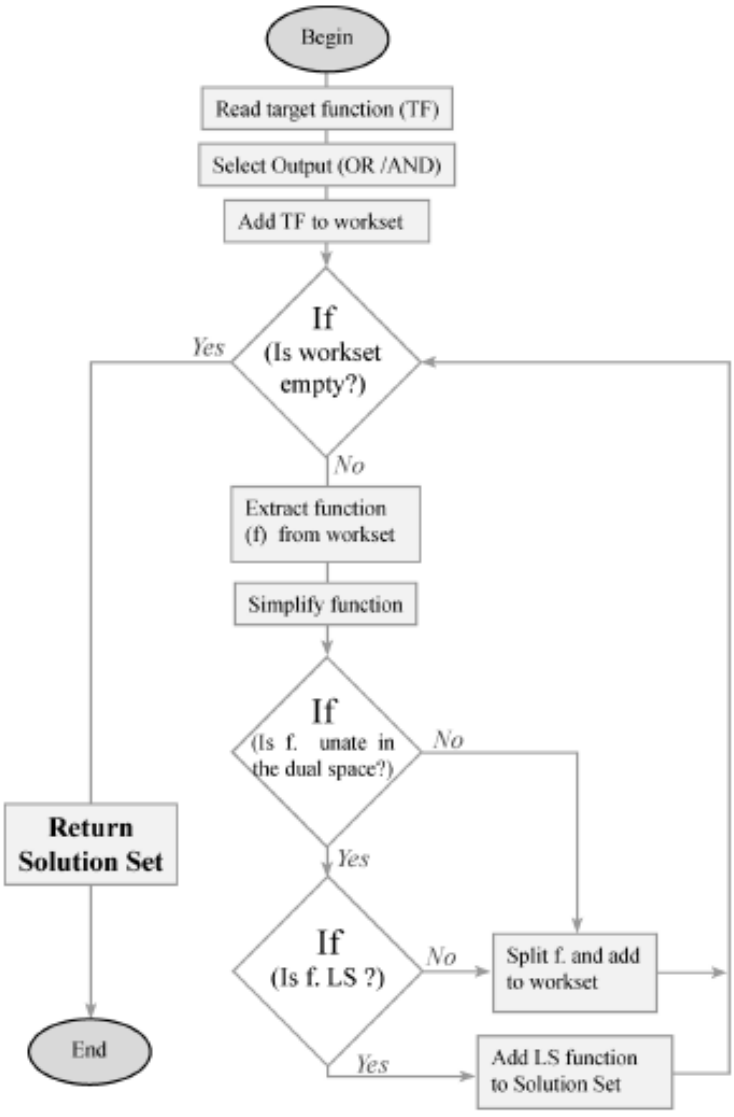


Figure 3.3: Truth table based synthesis approach proposed by Subirats *et. al* in (SUBIRATS; JEREZ; FRANCO, 2008).

In (GOWDA et al., 2011), GOWDA et al. use a factorized tree method to generate the network of threshold gates. The method recursively breaks the initial expression tree into sub expressions, identifying sub-trees which represent TLFs and assigning the input weights. It is more appropriate for IC synthesis using several TLGs, but it is time consuming and the result strongly depends on the initial structure, in particular, on the ordering of tree nodes.

The two methods presented by Gowda are based on efficient traversal of decision diagrams (binary decision diagrams - BDD and max literal factor trees - MLFT), which allows not only to quickly identify that a given function is a threshold function, but also to compute its minimal weight assignment. The basic threshold identification procedure is used to decompose a Boolean function into a network of threshold functions. It also allows for exercising limitations on certain gate parameters such as maximum fanin, sum of weights, and function threshold making it more suitable for synthesis of realizable threshold circuits. Fig. depicts an example of Gowda's method, identifying a given threshold function over a BDD structure.

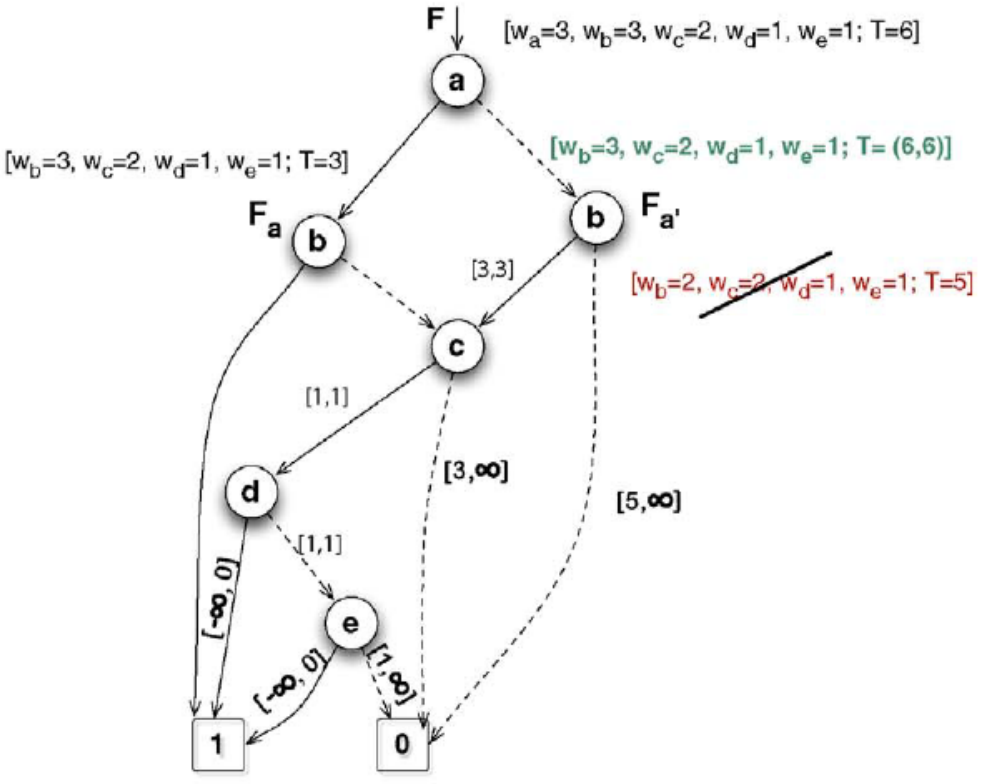


Figure 3.4: Example of the threshold synthesis approach proposed by Gowda *et al.* in (GOWDA et al., 2011).

The results demonstrate improvements in terms of number of gates when comparing to previous works. However, the authors do not present results in terms of circuit logic depth. Moreover, the method is slow and the results strongly depends on the BDD or MLFT variable ordering. Besides, the identification procedure is not effective, even for functions with smaller number of inputs.

The method proposed by Palaniswamy in (PALANISWAMY; TRAGOUDAS, 2014), implicitly implements Gowda's method (GOWDA et al., 2011). The work is based on a downhill approach which looks for circuit outputs that can be implemented in single TLGs. However, this work suffers from the same drawbacks of Gowda's method.

The method proposed by Neutzling, in (NEUTZLING et al., 2014), is based on a TLG association procedure through the principle called functional composition (MARTINS; RIBAS; REIS, 2012), based on dynamic programming. The algorithm associates simpler sub-solutions, with known costs, in order to produce a final solution with minimum cost.

Optimal TLG implementations containing all functions up to 4 inputs are generated with a straightforward procedure. This approach is interesting for a mapping point-of-view, since it is necessary only one execution to generate a full library, and the results are stored for posterior reuse, so avoiding the matching task.

For functions larger than 4 inputs a heuristic method based on the Boolean factoring algorithm presented by Martins *et al.*, in (MARTINS et al., 2010), is proposed. The first step for the synthesis of threshold network up to 6 inputs is to check if the target function is TLF. If this condition is attained, the algorithm returns a TLG provided by the identification algorithm. Otherwise, the heuristic algorithm starts by decomposing the target function in cofactors and cube cofactors, and these functions are stored in a set of derived functions. The next step is to discover the implementation of each function in this set, calling recursively the algorithm. After all functions in the set have an implementation in threshold network, the second step is combining these functions using AND/OR operations in order to generate new functions.

This method presents better results in terms of TLG count when compared to previous approaches. However, it does not present significant optimization in terms of logic depth. Moreover the algorithm does not scale for functions with a larger number of inputs.

The threshold logic synthesis methods, as mentioned above, generate a TLF network from a general Boolean function netlist. Another set of approaches focuses in the

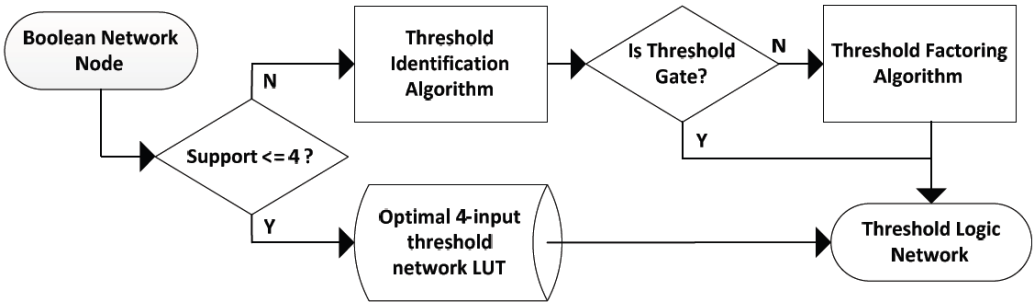


Figure 3.5: Threshold logic synthesis approach propose by Neutzling *et. al* in (NEUTZLING et al., 2014).

optimization starting from a TLF network. As consequence, they need to use one of the mentioned threshold logic synthesis approaches as input of their methods.

Methods for TLG based circuit rewiring are presented by Kuo *et al.*, in (KUO; WANG; HUANG, 2011), and by Lin *et al.*, in (LIN et al., 2014). Kuo's approach focuses only on circuit restructuring for satisfying a new fanin constraint and does not take into account the area or level minimization issue. Lin's approach, on the other hand, represents a heuristic for rewiring the circuit by minimizing the summation of input weights and function threshold value.

In (CHEN; WANG; CHANG, 2016), Chen *et al.* propose an analytic approach based on collapsing two threshold gates in order to minimize the total number of TLGs. This approach is different from all others because the method is performed directly over the AIG, without running a traditional threshold logic synthesis. The authors does not justify the greedy strategy and, although it is a recent work (2016), they compare the results only to the Zhang's approach (ZHANG et al., 2005).

Finally, Kulkarni *et al.*, in (KULKARNI et al., 2016), propose TLG-based approaches to reduce the circuit area and power consumption without loss of performance. However, the technology mapping is performed by a commercial tool using a conventional (non-threshold-based) standard cell library. This method replaces some conventional flip-flops by threshold logic sequential cells, latches and flip-flops (KULKARNI et al., 2016). As a consequence, a hybrid netlist comprising both TLGs and conventional logic gates is generated.

3.1 Thesis proposal

Notice that all of the mentioned threshold logic synthesis approaches focus basically on synthesizing single output non-TLFs. The first step of previous methods relies on a complete synthesis process which disregards threshold logic domain. Therefore, they do not explore the whole circuit where they would find, for instance, TLFs between different functions in the netlist.

In this thesis, we propose the first effective technology mapping approach for threshold logic gates, which is based on identifying threshold logic functions during the mapping. This enables to explore the entire circuit-level search space, seeking a threshold logic covering. As a consequence, we improve both area and performance results, as well as the synthesis scalability.

A second contribution introduced improves the quality of results by efficiently exploiting redundant cuts. Moreover, the technology mapper, we propose herein, is also able to target different threshold-based area estimations: the total summation of input weights and function threshold values; the total summation of gate inputs; or the total number of TLGs. Finally, the proposed technology mapping can be modified to handle majority (MAJ) functions, generating a MAJ gate netlist for different fanin restrictions.

4 PROPOSED THRESHOLD LOGIC SYNTHESIS

Given a digital circuit functionality, usually described in RTL format, the threshold logic synthesis (TLS) relies on finding an optimized threshold logic network. The most TLS tools initially cover the circuit by single-output nodes, disregarding their thresholdness, as illustrated in Fig. 4.1(a) (ZHANG et al., 2005; SUBIRATS; JEREZ; FRANCO, 2008; GOWDA et al., 2011; PALANISWAMY; TRAGOUDAS, 2014; NEUTZLING et al., 2014). In the next, these approaches perform the resynthesis of any non-TLF found to individual, unshared threshold logic network in order to provide the final covering.

In a preliminar approach, also presented in (NEUTZLING et al., 2015), we propose a synthesis flow that identifies TLFs before the circuit covering task. It is based on a three-stage procedure, as depicted in Fig. 4.1(b): (*i*) a complete cut enumeration, storing Boolean functions of cuts in the design; (*ii*) the identification of TLFs related to this set of computed cuts; and (*iii*) the technology mapping considering thresholdness of precomputed functions. By doing so, this approach is able to discard those non-TLFs and provides a threshold network since the first covering action. This strategy allows the exploration of multi-objective technology mapping algorithms. On the other hand, notice that this approach relies on computing cuts twice in the flow, in stages (*i*) and (*iii*) above.

In the following, we present our threshold logic synthesis method, depicted on Fig. 4.1(c), based on the LUT-based technology mapping. For this, we discuss the two main steps of the priority cut-based technology mapping algorithms: the cut enumeration and cut covering. Common area and delay cost functions for threshold logic synthesis are, respectively, TLG count and logic circuit depth. Thus, it is straightforward to relate threshold logic synthesis with FPGA technology mapping.

This section is organized as follows. Firstly, we present how the traditional cut enumeration works, presenting also two strategies to optimize this procedure. In Subsection 4.2, we propose an approach to identify threshold functions during the cut enumeration. Next, we discuss how we choose the best cuts, during the traditional covering. Finally, in Subsection 4.4, we present threshold logic covering based on priority cuts. Moreover, we present how to consider different TLG area estimations during the mapping, like the summation of input weights or the overall number of gate inputs.

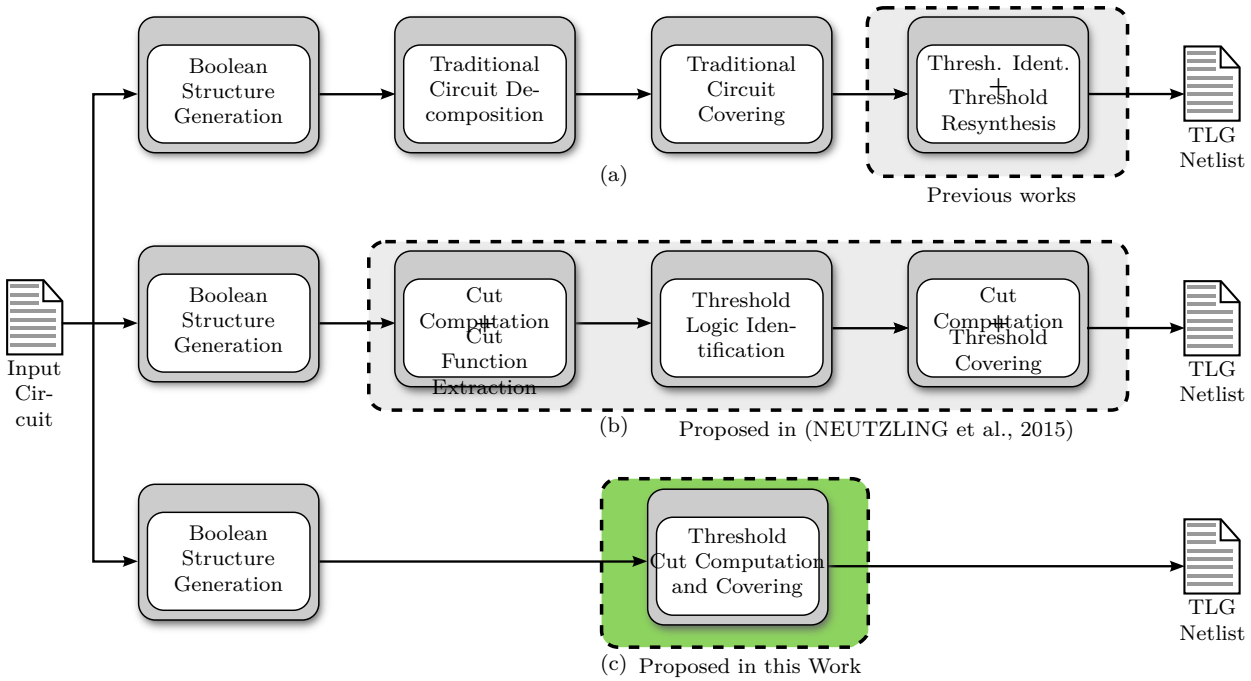


Figure 4.1: Comparison on the different threshold synthesis flow in the literature.

4.1 Traditional Cut Enumeration

Technology mapping transforms a technology-independent logic network, called the subject graph, into a network of logic nodes. In FPGAs, each logic node is represented using a K-input look-up table (LUT) implementing any Boolean function up to K inputs. The subject graph is often represented as an AIG comprising of two-input ANDs and inverters.

Most structural methods of FPGA mapping (CONG; DING, 1994; CHEN; CONG, 2004; MISHCHENKO; CHATTERJEE; BRAYTON, 2007; MISHCHENKO et al., 2007) start by computing all, or nearly all, K-feasible cuts for each AIG node. Similar methods exist for standard cell mapping. In the next, the AIG nodes are traversed in a topological order and a dynamic programming approach is used to find an optimum-depth LUT mapping of the AIG. In this section, we demonstrate the cut enumeration procedure.

As we defined in Section 2, a *cut* c of a node n is a set of nodes of the network, called *leaves* of the cut, such that every path between a PI and n contains a node in c. The cut enumeration is the task to compute a cut set for each node in the AIG.

Let A and B be two sets of cuts. We define the operation $A \bowtie B$ as follows:

$$A \bowtie B \equiv \{a \cup b \mid a \in A, b \in B, |a \cup b| \leq \mathcal{K}\}. \tag{4.1}$$

Let $\Phi(n)$ be the set of K-feasible cuts of node n. If n is an AND node, let n_1 and n_2 denote its fanins. We have

$$\Phi(n) \equiv \begin{cases} \{\{n\}\} & : n \in M_{1_{in}}; \\ \{\{n\}\} \cup \{\Phi(n_1) \bowtie \Phi(n_2)\} & : otherwise, \end{cases} \tag{4.2}$$

This formula translates into a simple procedure that computes all K-feasible cuts in a single pass from the PIs to the POs in a topological order. Informally, the cut set of an AND node is computed by merging the two cut sets of its children and adding the trivial cut (the node itself). This is done by taking the pairwise unions of cuts belonging to the fanins.

Figure 4.2 illustrates the bottom-up cut enumeration procedure for a small circuit.

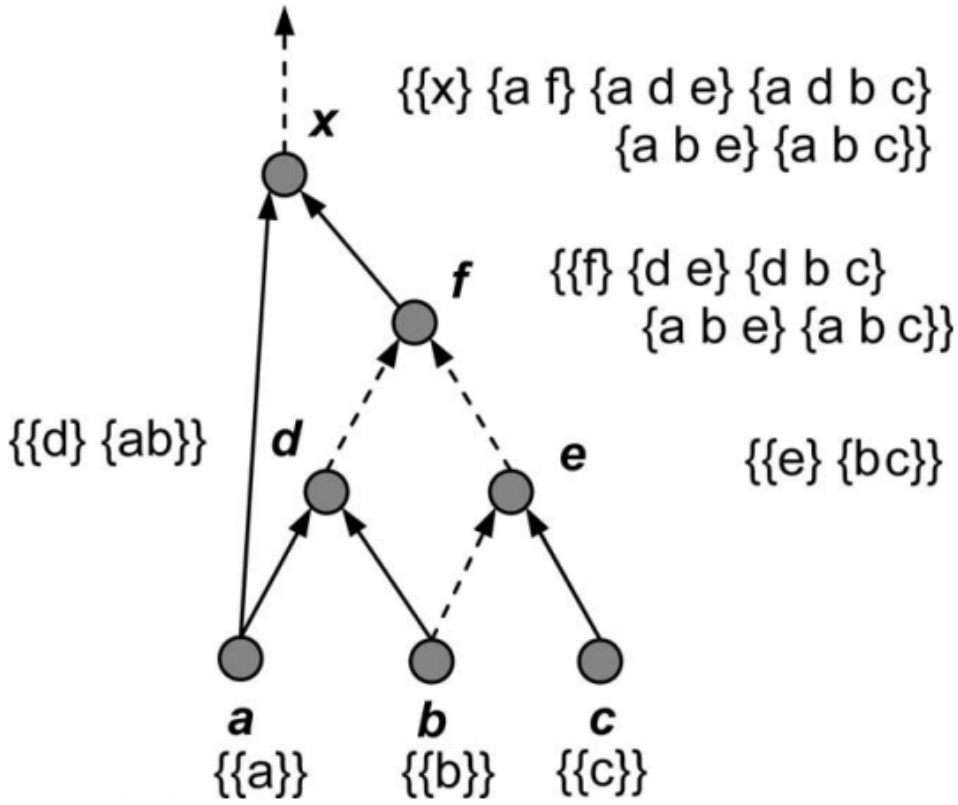


Figure 4.2: Example of cut enumeration.

4.1.1 Discarding cuts larger than K

In this process of merging the cut sets to form the resulting cut set, it is necessary to detect duplicated cuts and to remove dominated cuts. Removing them before computing cuts for the next node reduces the number of cut pairs considered without impacting the quality of mapping. In practice, the total number of cut pairs tried during the merging greatly exceeds the number of K-feasible cuts found.

Since the final mapping contains only LUTs up to Kinputs, any cuts larger than K-inputs can be pruned from the cut set. When merging cuts, the minimum number of inputs is determined by the largest cut. It was demonstrated in (PAN; LIN, 1998; CHEN; CONG, 2004) that discarding these cuts larger than K do not impact in the quality of results.

4.1.2 Discarding dominated cuts

A cut is said to be dominated if there is another cut of the same node, which is contained, set-theoretically, in the given cut. For instance, in Figure 4.2, the cutset of node x contains a dominated cut *adbc* (dominated by *abc*) that may be removed without affecting the quality of mapping.

In traditional state-of-the-art mappers for FPGA technology, dominated cuts are discarded during the cut enumeration (PAN; LIN, 1998; CHEN; CONG, 2004; MISHCHENKO; CHATTERJEE; BRAYTON, 2007). The methods applies a dominance checking for each operation $\{\Phi(n_1) \bowtie \Phi(n_2)\}$ so that dominated cuts are not considered.

4.2 Threshold Logic Cut enumeration

In a threshold logic technology mapping, we must guarantee that only the cuts that the corresponding function is TLF can be chosen in the final solution. In this section, we discuss how we efficiently determine if a structural cut is a threshold cut. Moreover, we demonstrate that obtaining the thresholdness information during the cut enumeration allows to improve the mapping results by exploring redundant cuts.

4.2.1 Identifying threshold cuts

Each structural cut represents a Boolean function. A threshold logic mapper needs to be aware which cuts represent TLFs. The *threshold logic identification* process verifies if a Boolean function is TLF (or not) and compute the respective input weights and the function threshold value. In this work, we have adopted the identification process presented in (NEUTZLING et al., 2017), instead of the integer-linear-programming-based (ILP) algorithms applied in previous works (ZHANG et al., 2005; SUBIRATS; JEREZ; FRANCO, 2008, 2008; GOWDA et al., 2011; PALANISWAMY; TRAGOUDAS, 2014). This is mainly due to fast runtime and good quality of results obtained.

This heuristic approach is a secondary contribution of this thesis, being better discussed in Section 6.2. Here, we present a summary of the results in order to justify the importance of such a method in the proposed flow:

- Our method scales better and recognizes more threshold functions.
- Our method is the only one that compares numerically to ILP, whereas other authors only state that ILP is slow, without providing numerical results.
- We have reassessed the comparison with the ILP-based method. Our heuristic is around 8 times faster than the ILP, missing only 1% of the threshold functions with up to 15 inputs present in large opencore benchmark circuits as sub-circuits.
- Our heuristic approach scales up to 92 inputs.
- For large functions, the proposed method is around 3 orders of magnitude faster than ILP.

4.2.2 Unique TL identification per NPN representative cut function

Although the identification we have adopted is very fast, such a procedure still can be a bottleneck due to the huge number of cuts created during the enumeration.

An effective solution for this issue is avoid repeating the identification for the same Boolean function. Actually, it can be even better if the identification is performed once for each NPN representative class, since the input weights and the function threshold value have the same magnitude for functions in the same NPN class (MUROGA, 1971).

In order to acomplish this strategy, two tasks are required:

1. Find as fast as possible an NPN representative for a given Boolean function;
2. Store the thresholdness information in a hashtable, defining the NPN representative as a hashtable key.

In this work, we adapt a software package called DSD manager to perform the above mentioned tasks. DSD manager was proposed in (MISHCHENKO; BRAYTON, 2007) for storing Boolean functions using their DSD structures, performing Boolean operations, and caching intermediate computations.

Disjoint-support decomposition (DSD) represents a completely-specified Boolean function as a tree of nodes with non-overlapping supports and inverters as optional complemented attributes on the edges (BERTACCO; DAMIANI, 1997; CALLEGARO et al., 2015). This tree is called the DSD structure of a function. Unlike truth tables and BDDs, DSD structures have the advantage of making Boolean properties of the function explicit. Several examples are:

- symmetric variables of a full DSD structure are found by detecting symmetric nodes (AND/XOR) at the leaves of the structure;
- decomposability of a full DSD into a network of K-LUTs is checked by computing K-feasible cuts, without an expensive search for a K-feasible bound-set;
- a DSD structure can be made canonical by permuting branches according to some ordering principle, and by removing complemented attributes from the inputs and the output, resulting in an NPN canonical form.

In our case, the NPN canonical form is very useful. Since the decomposition is very fast (orders of magnitude faster than the threshold logic identification procedure),

for each new cut, we find the corresponding DSD structure and check in the hash table if the NPN representative of such a DSD structure was already identified. If not, we perform the identification and store in the hash the thresholdness information.

The structure developed to hash the thresholdness information of the cuts is quite efficient, allowing the proposed approach scales even for large circuits. The results in Section 5 present threshold synthesis for circuits containing more than twenty millions of AIG nodes.

An overview of the threshold logic cut enumeration step is presented in Algorithm 4.1. The idea of find the NPN representative of the cut function is performed in line 9. The hashing is performed between lines 10 and 15. The procedure of cut sorting in lines 16-20 is part of the covering step and will be properly explained in Section 4.3 and in Section 4.4.

Algorithm 4.1: Pseudo-code of the proposed approach.

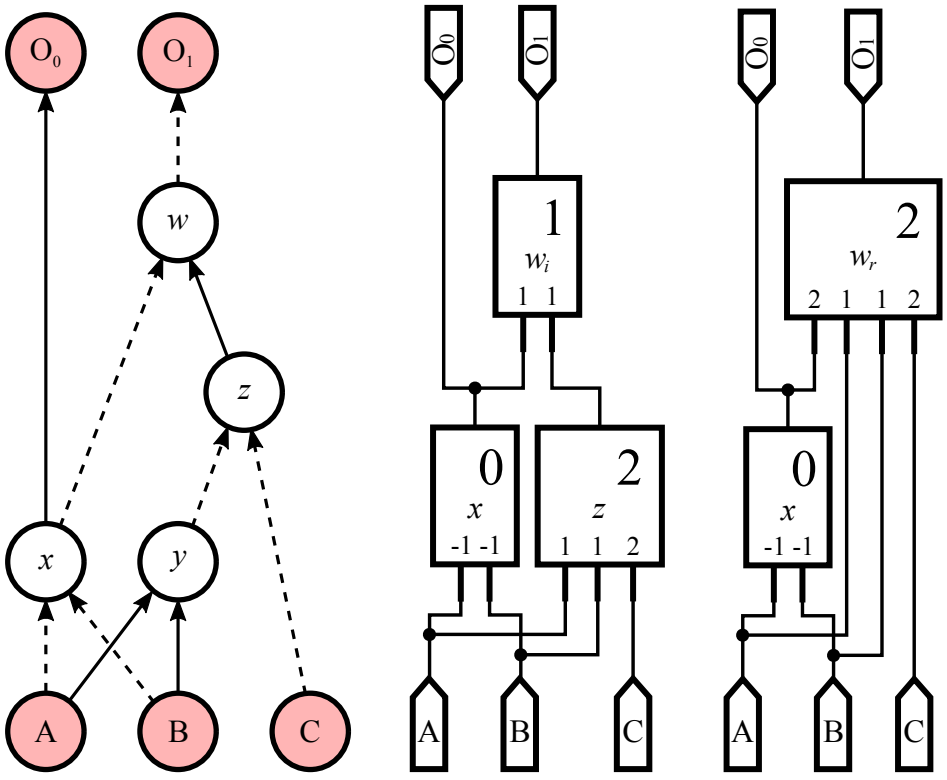
```
   Input: AIG, K, C
   Output: AIG with covering information
   /* traverse and nodes in topo order                                  */
 1 foreach node ∈ AIG.andNodes do
 2     cutSet = empty cut set;
 3     foreach cut0 ∈ node.fanin0.cutSet do
 4         foreach cut1 ∈ node.fanin1.cutSet do
 5             cut = merge(cut0,cut1);
 6             if cut.nLeaves > K then
 7                 continue;
 8             cutFunc = extracted func. from cut;
 9             cutNpn = NPN represent. of cutFunc;
10             if cutNpn is hashed then
11                 cut.isThreshold = get from hash cut.thresholdArea = get from
                    hash;
12             else
13                 cut.isThreshold = run identification;
14                 cut.threshArea = get from identification;
15                 add this information into the hash;
16             add cut into cutSet and sort it;
17             if cutSet.nCuts > C then
18                 discard the worse cut ∈ cutSet;
19     add the trivial cut into cutSet and sort it;
20     node.bestCut = best threshold cut in CutSet;
```

4.2.3 Improving Quality-of-Results by Enumerating Redundant Cuts

In the context of EDA and logic synthesis, the state-of-the-art algorithms for cut computation rely on enumerating irredundant cuts only. By pruning redundant cuts, the enumeration methods have a better performance (both in terms of runtime and memory usage) with almost no losses on the QoR for the general cases (near-optimal logic depth is still achievable, for instance). This procedure is taken as a ground base and it is implemented in the main state-of-the-art methods for cut computation, such as the priority cuts approach (MISHCHENKO et al., 2007).

In this work, we demonstrate that cleverly relaxing the cut redundancy checking can decrease area in the final TLG nestlist without a significant impact neither on the circuit delay nor on the methods performance. The following example, depicted in Figure 4.3, illustrates how this QoR improvement is achieved.

Consider the case in which the AIG shown in Figure 4.2(a) is taken as input in a threshold mapper. If only irredundant cuts are enumerated, the most likely covering



(a) (b) (c)

Figure 4.3: Example of area reduction by enumerating redundant cuts: a given AIG (a); mapped circuit by enumerating only irredundant cuts (b); mapped circuit with reduced area by enumerating redundant cuts (c).

has three TLGs in the mapped circuit, as illustrated in Figure 4.2(b). The cuts used in this covering are $x = \{A, B\}$, $z = \{A, B, C\}$ and $w_i = \{x, z\}$. Now, if we allow for redundant cuts to be additionally enumerated, a covering with only two TLGs is achievable, as depicted in Figure 4.2(c). In this case, the cuts used in the covering are $x = \{A, B\}$ and $w_r = \{x, A, B, C\}$. Notice that the cut w_r would never be enumerated in the prior mapping, since it is redundant with cut $z = \{A, B, C\}$ (as w_r is a superset of z). However, computing the redundant cut w_r allowed for decreasing the number of TLGs in the mapping from three to two without impacting the circuit delay.

The idea of looking for redundant threshold cuts was primarily cited by KULKARNI; VRUDHULA, in (KULKARNI; VRUDHULA, 2016). In that work, the authors claim that, despite some threshold cuts are not enumerated for being redundant, including redundant cuts in the enumeration procedure would have large computational requirements. To overcome this issue, they propose (a) to change from node cut to line cut computation, and (b) to compute only unidirectional cuts.

In order to avoid such a change on the cut computation paradigm, but still trying to keep the procedure computationally feasible, we propose to relax the cut redundancy checking while enumerating priority cuts. Thus, we allow for redundant threshold cuts to be added in the priority cut set (according to the sorting function). Such a strategy is possible since the thresholdness identification is performed during the cut enumeration. The experimental results presented in Section 5 demonstrate that such a proposed strategy provides improvments in the quality of results with a bearable execution time and memory usage.

4.3 Traditional Cut Covering

The conventional algorithm for delay optimal K-LUT mapping enumerates all K-feasible cuts and chooses the best set of cuts by using dynamic programming on the AIG. The algorithm proceeds in two passes over the nodes. The first pass, called the forward pass, is in topological order from PIs to POs. For each node, all K-feasible cuts are enumerated (using the K-feasible cuts of its children), and the cut with the earliest arrival time is selected.

The second pass of the algorithm is done in reverse topological order. For each PO, the best K-feasible cut is chosen, and an LUT is constructed in the mapped netlist to implement it. Then, recursively for each node in this best cut, this procedure is repeated.

State-of-the-art FPGA technology mappers (CHEN; CONG, 2004; CONG; DING, 1994; MISHCHENKO; CHATTERJEE; BRAYTON, 2007) produce near-optimum logic depth while minimizing the number of LUTs in the resulting network. A typical procedure consists of the following steps:

1. Near-optimum delay mapping: compute arrival time at each node by computing the depth of priority cuts and choosing the best one;

2. Area recovery: improve area using several heuristics (for instance, area flow and exact local area (PAN; LIN, 1998));

3. Choose the resulting cover.

4.3.1 Delay-oriented covering

The delay of a FPGA circuit is determined by two factors: (1) the delay in K-LUTs; and (2) the interconnection delay. Each K-LUT has a constant delay independent of the function it implements (the access time of a K-LUT). The interconnection delay is dominated by the physical configuration of the FPGA, which is not available during synthesis. Therefore, the state-of-the-art FPGA technology mappers assume that each edge in the mapping solution has a constant delay. Due to these reasons, the circuit delay is commonly represented by the logic depth and the circuit area is represented by the number of K-LUTs in the mapping solution (PAN; LIN, 1998; CHEN; CONG, 2004; MISHCHENKO; CHATTERJEE; BRAYTON, 2007).

Algorithm 4.2 shows the pseudo-code of depth-oriented mapping performed in the traditional FPGA mapping.

Algorithm 4.2: Depth-oriented traditional FPGA covering.

```
   Input: AIG, K
   Output: AIG with covering information
   /* traverse and nodes in topo order                                  */
1  foreach node ∈ AIG.andNodes do
2      bestCut=NULL;
3      foreach cutC ∈ node.cutSet do
4          if bestCut=NULL OR getLevel(bestCut) > getLevel(C) then
5              bestCut=C;
6      setLevel(n,getLevel(bestcut));
7      setRepresentative(n,bestcut);
```

The AIG nodes are considered in a topological order. At each node, all cuts are enumerated and an optimum-depth cut is found. This cut along with its level is stored at the node. The level of a cut is computed by adding 1 to the largest level of the cut fanins, *i.e.*,

$$level(C) = 1 + maxlevel(n) \tag{4.3}$$

where $n \in C$ and $level(n)$ is the best level for node n (from among all its K-feasible cuts). This recursion is well defined, since when the cuts for a node n are being processed, the nodes in the transitive fanin of n have already been processed. Thus, the best arrival time of any node in a K-feasible cut of n has already been computed.

4.3.2 Area-oriented covering

Next to find a near-optimum mapping solution in terms of delay, a heuristic is applied in order to improve the area. The heuristic explored in this work, called *area flow*, has a global view and selects logic cones with more shared logic.

Area flow, presented in (MANOHARARAJAH; BROWN; VRANESIC, 2006), *a.k.a.* effective area, is a useful extension of the notion of area. It can be computed

in one pass over the network from the PIs to the POs. Area flow for the PIs is set to 0. Area flow at a node n is the following:

$$AF(n) = [Area(n) + \sum_i AF(Leaf_i(n))]/NumFanouts(n) \quad (4.4)$$

where Area(n) is the number of LUTs needed to map the current best cut of node n. Area(n) can be 1 or larger if some of the fanins of the top most LUT do not have external fanouts. Leaf$_i$ (n) is the i-th leaf of the best cut at n, and NumFanouts(n) is the number of fanouts of node n in the currently selected mapping. If a node is not used in the current mapping, for the purposes of area flow computation, its fanout count is assumed to be 1.

If nodes are processed from the PIs to the POs, computing area flow is fast. Area flow gives a global view of how useful the logic is in the cone for the current mapping. Area flow estimates sharing between cones without the need to re-traverse them.

4.3.3 Choose the resulting cover

The procedure used in the traditional mapping to derive the final LUT network is shown in Algorithm 4.3. The procedure assumes that one K-feasible representative cut is assigned at each node. Two sets of AIG nodes are supported: the nodes used in the mapping (M) and the nodes currently present in the frontier (F). While the frontier is not empty, one node (n) is extracted from it, added to the mapping, the representative cut of this node is computed, and the leaves of this cut are explored. If a leaf of n does not belong to the mapping or is not a PI, this leaf is added to both the mapping and the frontier. When the frontier is empty, the procedure terminates and returns the set M of nodes used in the mapping. Each of these nodes will be implemented by a LUT.

Algorithm 4.3: Producing the mapped LUT network.

```
Input: AIG, K
Output: set of nodes mapped containing the nodes used in the final mapping
1  mapped=NULL;
2  frontier=NULL;
3  while frontier ≠ ∅ do
4      n=extractNode(frontier);
5      insertNode(n,mapped);
6      cut=getCut(n);
7      foreach cutnode m ∈ cut do
8          if m ∉ mapped OR m ∉ PIs then
9              frontier=frontier ∪ m;
10 return mapped;
```

4.3.4 Priority cuts

As discussed before, the traditional approach for delay optimal K-LUT mapping enumerates all K-feasible cuts and chooses the best set of cuts by using dynamic programming on the AIG. The main limitation of the conventional algorithm is that it explicitly enumerates a large number of all K-feasible cuts during the forward passing.

The *priority cuts* approach avoids exhaustive cut enumeration by computing only a small fixed number (typically, 5-10) of "good" K-feasible cuts at each node. The criteria used to prioritize the cuts differ depending on the mapping goals. The candidate cuts are sorted using a sorting function, and the best C cuts are found and stored at the node. In practice, sorting is done on the fly by keeping only the C best cuts at any time. Finally, the best cut from the previous pass is always added to the set of at most $(C+1)^2$ candidate cuts derived using the fanins cuts. As a result, the mapping procedure never loses good cuts. If this is not done, the best cut may be lost due to the heuristic nature of cut selection.

Algorithm 4.4 represents the cut sorting procedure. The command *isBetterThan* compare two cuts according to a defined sorting function. Different sorting functions are used at each mapping pass. If there is a tie, a secondary cost function is compared.

Experiments indicate that such a prioritization gives a depth-optimum mapping for 95% of all benchmarks and LUT sizes, even if only one cut is stored at each node. Storing 8 priority cuts per node allows the algorithm to avoid area penalty due to not enumerating all cuts. For 6-input LUTs, memory is reduced 10x and runtime 5x compared to previous approaches, while circuit depth and area are comparable or better. For 8-input and larger

LUTs, the reduction in memory and runtime is about 50x (MISHCHENKO et al., 2007) .

Algorithm 4.4: Cut sorting procedure.

```
Input: Set of cuts CutSet, Cut current
Output: Sorted et of cuts CutSet
/* traverse the cutset from the worse cut to the best one      */
1 foreach cut ∈ cutSet do
2     if current isBetterThan cut then
3         SWAP(current,cut);
4     else
5         return;
```

4.4 Threshold cut covering

In our proposed approach to perform threshold logic synthesis, we explore the priority cut based technology mapping. The main objective in the covering step is selecting only threshold cuts. This is possible since we have performed the identification procedure during the enumeration. In Subsection 4.4.1, we present different strategies for sorting the cuts during the covering.

When performing the identification during the enumeration, besides to know if the cut represents a threshold logic function, we compute the input weights and the function threshold value required to implement such a function. In Subsection 4.4.2, we demonstrate how we use these information in order to optimize different area estimations in the proposed method.

4.4.1 Threshold cut sorting

In our threshold logic technology mapping approach, we propose different cut sorting strategies, based on the cut thresholdness. At first, we guarantee that the best cut is always a threshold cut, *i.e.*, the function implemented by the cut can be implemented in a single TLG. This strategy results in netlist composed only by TLG.

For this, we modify the cut sorting procedure. Before starting the traditional cut sorting, we check if the best cut is not threshold and the current cut is threshold. In

this case, we exchange these two cuts. Besides, if the best cut is threshold, it can not be replaced by a non-threshold cut. Such a modification procedure is presented in Algorithm 4.5.

Algorithm 4.5: Threshold cut sorting procedure.

```
   Input: Set of cuts CutSet, Cut current
   Output: Sorted set of cuts CutSet
1  if current.thresholdness AND !bestCut.thresholdness then
2      SWAP(current,bestCut);
3      cutSorting(CutSet,cut);
4      return;
   /* traverse the cutset from the worse cut to the best one      */
5  foreach cut ∈ cutSet do
6      if current isBetterThan cut then
7          SWAP(current,cut);
8      else
9          return;
```

Although we restrict the best cut to threshold cuts, the other cuts stored at the cut set for each node can be non-threshold. An alternative strategy is to increase the number of threshold cuts in the cutset, assuming that threshold cuts can potentially generate other threshold cuts. While non-threshold cuts are generating non-threshold cuts.

Based on this assumption, we proposes 4 different priority cut sorting approaches, focusing on privileging threshold cuts.The first strategy is prioritizing threshold cuts in the cut set. Such a strategy would potentially increase the number of threshold cuts in the next levels. Non-threshold cuts stay in the cutset only if the number of threshold cuts is less than the cutset capacity.

In order to implement this cut sorting strategy, we have modified the function *isBetterThan* in Algorithm 4.5 which is responsible to compare two cuts. Before to compare by using the defined cost function, *i.e.*, delay, area or inputs, the method compare the thresholdness of the cuts. A threshold cut is always considered better than a non-threshold cut and the other cost functions are used as tie-brakers.

All threshold functions are unate, but there are unate functions which are not threshold. An interesting issue is the potential of non-threshold unate cuts to generate threshold cuts. For instance, given two non-TLF cuts C_1 and C_2 that implement the functions $F_1 = ab+cd$ and $F_2 = ac+bd$, that are non-TLF, the resulting cut C_3 implements

the function $F_3 = a(bc + bd + cd) + bcd$, which is TLF. Therefore, the second strategy allows unate cuts at the cut set.

Another possibility is that binate functions can generate good threshold functions. Taking it into account, we also proposes an strategy which increases the number of threshold (unate) function without discarding the best non-TLF cuts. We do this reserving at least half of the cut set for the TLF cuts and half for non-TLF cuts.

Finally, we experiment the same strategy for cuts which implement unate functions. We reserve at least half of the cut set for the unate cuts and half for the other cuts.

Thee summary of the threshold logic priority cuts strategies are listed in the following and illustrated in Figure 4.4.

- (a) Best cut Threshold:
- (b) All cuts Threshold:
- (c) All cuts Unate:
- (d) Half cut set Threshold:
- (e) Half cut set Unate:

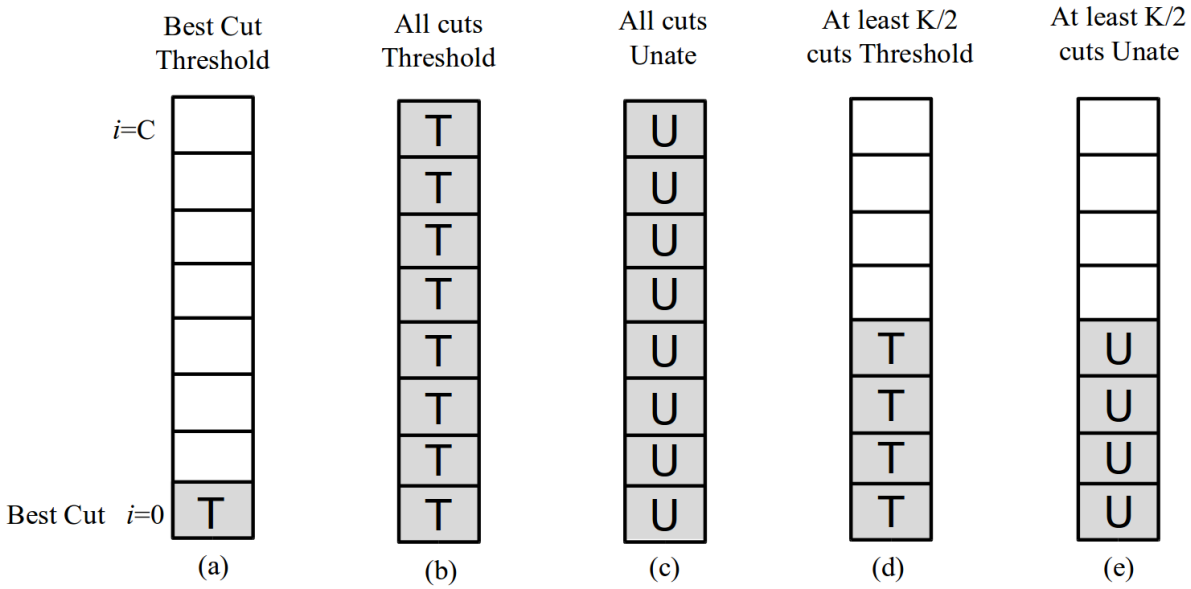


Figure 4.4: Different cut sorting strategies for the threshold priority cut approach.

4.4.2 Different TLG area estimations

To the best of our knowledge, the state-of-the-art works on threshold logic synthesis are limited to optimize the overall number of TLGs. Although two of them present numbers regarding different threshold-based area estimations, such works still optimize the total number of TLGs during the synthesis. ZHANG et al. in (ZHANG et al., 2005), just reports those numbers in the final netlists. KULKARNI et al., in (KULKARNI et al., 2016), uses the Zhang's synthesis method to obtain a TLG netlist and proposes a rewiring method to locally improve the summation of input weights and function threshold value.

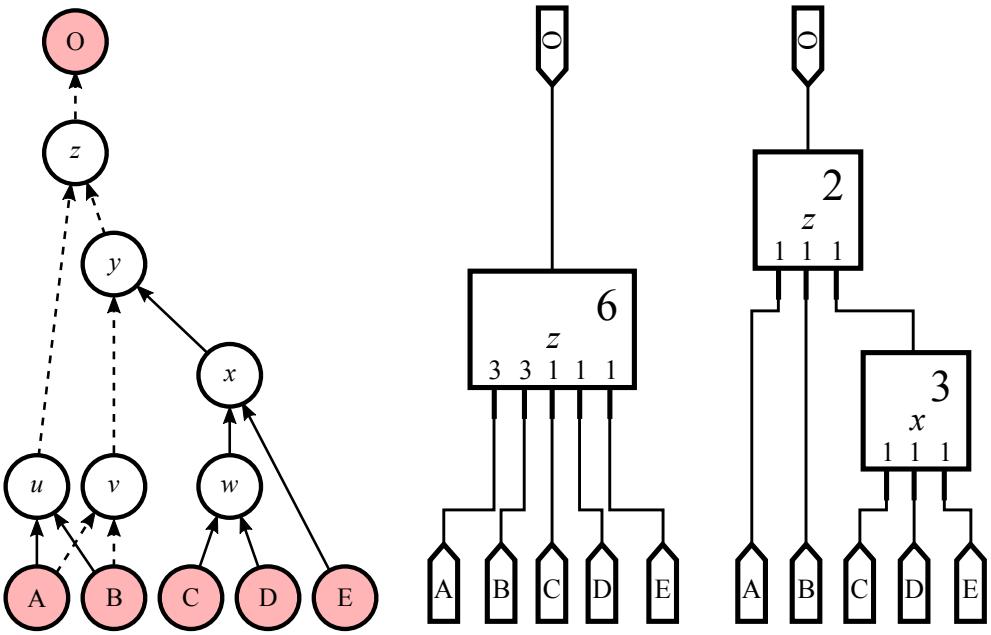
We propose an effective technology mapping which is able to optimize different threshold-based area estimations. The considered area estimations are either (*i*) the overall number of TLGs, (*ii*) the summation of input weights and threshold values, and (*iii*) the overall summation of gate inputs. Such area estimations are more suitable with the state-of-the-art TLG physical implementations and were discussed in Section 2.2.6.

The following example demonstrates an instance where optimizing the suitable cost function impacts on different mapping results. Given the circuit represented by the AIG in Fig 4.4(a), the whole circuit can be implemented in a single 5-inputs TLG. Such a solution is optimal in terms of TLG count. However, it is not the best solution if the area estimation is defined by the overall summation of input weights and function threshold value. Considering this area estimation, the solution with one TLG, presented in Figure 4.5(b), has the total area equals to 15. In contrast, the best solution, showed in Figure 4.5(c), is obtained with two TLGs and the total area is equal to 11. Such a solution is found by the proposed method.

In order to optimize different area estimations, we compute the TLG area along with the threshold identification procedure. As discussed in Section 4.2, the threshold identification is performed only once per NPN representative during the cut enumeration. If the identified function is threshold, the corresponding TLG area is computed and stored in the hash table. Afterwards, such an information is used during the covering step.

This TLG information are used more specifically during the area recovery step, replacing the LUT area in the area flow equation. As a result, the area flow is computed asfollows:

$$AF(n) = [TLGArea(n) + \sum_{i} AF(Leaf_i(n))]/NumFanouts(n). \tag{4.5}$$



(a) (b) (c)

Figure 4.5: Example of different TLG netlists obtained by optimizing different area estimations: a given AIG (a); mapped circuit optimizing the overall number of TLGs (b); mapped circuit optimizing the summation of input weights and function threshold values (c).

5 EXPERIMENTAL RESULTS

In order to validate the proposed threshold logic synthesis approach, experiments were carried out taking into account three different sets of benchmark circuits. The first set presents results to support the claims stated in Section 4. In the second set, the proposed work is compared to related threshold logic synthesis approaches from other authors (ZHANG et al., 2005; SUBIRATS; JEREZ; FRANCO, 2008; GOWDA et al., 2011; PALANISWAMY; TRAGOUDAS, 2014; NEUTZLING et al., 2014; LIN et al., 2014; CHEN; WANG; CHANG, 2016). In the third set of experiments, we present a comprehensive collection of results when customizing the developed mapper to achieve different mapping goals (*i.e.*, circuit area versus logic depth) and targeting different threshold-based circuit area estimations.

All results have been checked through the combinational equivalency by using the ABC "*cec*" command (Berkeley Logic Synthesis and Verification Group, 2017). For the sake of reproducibility, all the experiments can be repeated using the complete dataset available in (NEUTZLING et al., 2017), which includes the optimized benchmarks and threshold synthesis scripts. The proposed approach is implemented in the ABC tool (Berkeley Logic Synthesis and Verification Group, 2017), using C programming language and compiled using *gcc 4.7.2*. The experiments were performed on a computer with Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz, 8GB RAM.

5.1 Evaluation of Proposed Contributions

First of all, we compare the new proposed unified flow for threshold logic synthesis, which **identify threshold cut during the enumeration**, with the previous work presented in (NEUTZLING et al., 2015), both based on cut pruning as discussed in Section 4. Opencore circuits are used as benchmarks (PISTORIUS et al., 2007). Table 5.1 presents the results obtained. Columns 2-7 refer to the method described in (NEUTZLING et al., 2015), comprising the three steps illustrated in Fig. 4.1(b), and present the mapping results (number of TLGs and circuit logic depth) followed by the execution time of each stage. Columns 8-10 refer to the single-step mapping proposed in this work. The execution time reduction is up to 53%, being 36% on average, so demonstrating the advantage on identifying threshold functions during the cut enumeration task. The new threshold synthesis flow improves both the execution time and the quality of results in terms of

Table 5.1: Summary of results for the proposed threshold logic synthesis approach in comparison to (NEUTZLING et al., 2015).

	Approach presented in (NEUTZLING et al., 2015)						Proposed work		
circuit	*#TLGs*	*logic depth*	*step 1 (sec)*	*step 2 (sec)*	*step 3 (sec)*	*total time (sec)*	*#TLGs (ratio)*	*logic depth (ratio)*	*total time (ratio)*
oc_aquarius	9271	29	16.77	0.81	17.10	34.68	(0.99)	(1.00)	(0.63)
oc_cfft_1024x12	4002	8	6.28	0.86	6.90	14.04	(0.99)	(1.00)	(0.76)
oc_cordic_p2r	4007	8	5.90	0.95	6.82	13.67	(0.98)	(1.00)	(0.73)
oc_cordic_r2p	4943	8	6.58	1.00	7.12	14.70	(0.98)	(1.00)	(0.74)
oc_des_perf	9245	7	26.79	0.19	27.97	54.95	(0.99)	(1.00)	(0.79)
oc_ethernet	3741	7	3.96	0.16	4.24	8.36	(0.99)	(1.00)	(0.61)
oc_fpu	8501	281	19.44	1.30	18.31	39.05	(0.96)	(1.00)	(0.66)
oc_mem_ctrl	6626	8	5.97	0.08	6.59	12.64	(0.99)	(1.00)	(0.59)
oc_video_dct	14748	13	34.22	1.24	32.36	67.82	(0.99)	(1.00)	(0.62)
oc_video_jpeg	19218	12	31.58	1.19	31.58	64.35	(0.98)	(1.00)	(0.66)
radar20	30745	14	73.96	2.12	65.38	141.46	(0.99)	(1.00)	(0.47)
uoft_raytracer	61879	23	210.75	3.17	182.54	396.46	(0.98)	(1.00)	(0.49)
geomean							(0.98)	(1.00)	(0.64)

TLG counting, without loss on circuit logic depth.

The benefits of performing threshold logic synthesis while **relaxing the cut redundancy checking** during the priority cuts enumeration is also demonstrated taking into account the opencore circuits as benchmarks (PISTORIUS et al., 2007). Table 5.2 presents the results in terms of the number of TLGs and the circuit logic depth on the final netlist, along with the runtime and memory usage during the execution. Columns 2-5 refer to the traditional cut enumeration, pruning the redundant cuts. Columns 6-9 refer to the proposed approach in which redundant cuts are enumerated during threshold logic synthesis. The results show that the proposed strategy allows for circuit area reduction up to 21%, being 11% on average, with no significant impact on the circuit logic depth. Although there are runtime and memory overheads, none of the circuit synthesis has taken longer than 4 minutes nor needed more than 130 MB of memory usage during mapping.

It is expected that **different area estimations** may deliver completely different mapped circuits. Table 5.3 compares the obtained results when the proposed mapper addresses the standard cost function (*i.e.*, the total number of TLGs) to the two new cost functions: the total sum of input weights and threshold values, and the total number of gate inputs. Columns 2-4 present the total sum of input weights and threshold values obtained from the two cost functions. Columns 5-7 show the total number of gate inputs when targeting the evaluated cost functions. Notice that the best results are obtained when the most appropriate area estimation is considered during the mapping. Regarding

Table 5.2: Trade-off obtained when relaxing the cut redundancy checking.

	Pruning Redundant Cuts				*Enumerating Redundant Cuts*			
circuit	*#TLGs*	*logic depth*	*run-time (sec)*	*memory (MB)*	*#TLGs (ratio)*	*logic depth (diff)*	*run-time (ratio)*	*memory (ratio)*
oc_aquarius	9426	29	20.55	20.62	(0.97)	0	(1.09)	(1.22)
oc_cfft_1024x12	4997	8	6.51	6.81	(0.79)	0	(1.70)	(3.46)
oc_cordic_p2r	4763	7	6.15	10.12	(0.83)	+1	(1.66)	(2.42)
oc_cordic_r2p	5752	7	6.95	10.27	(0.84)	+1	(1.63)	(2.32)
oc_des_perf	10415	7	26.59	49.39	(0.88)	0	(1.72)	(2.16)
oc_ethernet	3774	7	4.79	5.48	(0.98)	0	(1.11)	(1.15)
oc_fpu	9345	281	21.10	24.29	(0.87)	+1	(1.24)	(1.98)
oc_mem_ctrl	6642	9	7.44	3.70	(0.99)	-1	(1.08)	(1.57)
oc_video_dct	17075	14	35.76	51.22	(0.86)	-1	(1.22)	(2.17)
oc_video_jpeg	20965	13	35.62	49.80	(0.90)	-1	(1.25)	(2.26)
radar20	34254	14	54.90	44.63	(0.89)	0	(1.26)	(2.02)
uoft_raytracer	70263	23	168.61	87.34	(0.86)	0	(1.18)	(1.39)
geomean					(0.89)		(1.32)	(1.92)

the total sum of input weights and threshold values, the overhead can be up to 88%, being 60% on average, when minimizing the number of TLGs. Concerning the total number of gate inputs, the results can be up to 19% worse, being 10% on average, whether the mapper optimizes the number of TLGs instead of the considered cost function. Therefore, a threshold logic mapper capable of optimizing circuits taking into account different area estimations tends to be more effective when targeting different technologies.

Finally, we evaluate the results obtained by the different **threshold cut sorting** approaches: (1) the best cut threshold, (2) all cuts threshold, (3) all cuts unate, (4) half of cuts threshold and (5) half of cuts unate. Table 5.4 present the results obtained in terms of number of gates and circuit logic depth. First column present the circuit names. Column 2 present the number of TLGs and columns 2-6 present presents the ratio of each

Table 5.3: Optimization for different TLG area estimations.

	area estimation ($\sum$W + T)			area estimation ($\sum$#inputs)		
circuit	CF=$\sum$W + T	CF=#TLGs	(ratio)	CF= $\sum$#inputs	CF=#TLGs	(ratio)
oc_aquarius	64776	97130	(1.50)	28372	32908	(1.16)
oc_cfft_1024x12	22949	42464	(1.85)	12826	14662	(1.14)
oc_cordic_p2r	27761	39244	(1.41)	12370	13310	(1.08)
oc_cordic_r2p	32236	50198	(1.56)	15455	16733	(1.08)
oc_des_perf	58212	93687	(1.61)	29378	30307	(1.03)
oc_ethernet	23349	35979	(1.54)	11695	12974	(1.11)
oc_fpu	43445	58186	(1.34)	23374	24269	(1.04)
oc_mem_ctrl	34363	54483	(1.59)	19684	21156	(1.07)
oc_video_dct	98118	175844	(1.79)	45729	54351	(1.19)
oc_video_jpeg	123335	232028	(1.88)	60407	68898	(1.14)
radar20	161436	241578	(1.50)	86378	94799	(1.10)
uoft_raytracer	355222	593636	(1.67)	191468	210825	(1.10)
geomean			(1.60)			(1.10)

Table 5.4: Evaluating different threshold cut sorting approaches.

	Number of TLGs					Logic depth				
circuit	sort1	sort2 (ratio)	sort3 (ratio)	sort4 (ratio)	sort5 (ratio)	sort1	sort2 (ratio)	sort3 (ratio)	sort4 (ratio)	sort5 (ratio)
cord1	4001	0.96	0.96	0.96	0.96	8	1.00	0.88	1.00	1.00
video	14711	0.99	0.99	1.02	1.00	13	1.00	1.00	1.00	1.00
cfft	3990	1.00	0.99	0.99	0.99	8	1.00	1.00	1.00	1.00
cord2	4932	0.98	0.98	0.99	0.98	8	1.00	1.00	1.00	1.00
aqua	9261	1.00	1.00	1.01	1.00	29	0.97	0.97	0.97	0.97
jpeg	19214	0.99	0.99	1.00	0.99	12	1.00	1.00	1.00	1.00
mem	6631	0.99	0.99	1.00	0.99	8	1.00	1.00	1.00	1.00
desperf	9219	0.95	0.96	0.96	0.96	7	1.00	1.00	1.00	1.00
fpu	8583	1.06	1.01	1.09	1.06	393	1.01	0.97	0.70	0.72
ether	3752	1.00	0.99	1.00	0.99	7	1.00	1.00	1.00	1.00
geomean		0.99	0.99	1.00	0.99		1.00	0.98	0.96	0.96

approach (1-4) related to approach number 1. Similarly, column 7 presents the circuit logic depth when using the approach number 1 and columns 8-11 presented the ratio when using the other approaches.

The results show that the difference is less than 5% in the most of cases and also in the geometric average. We think that, although we increase the total number of enumerated threshold cuts, the best cuts are the same for all approaches. For the FPU (floating point unit) circuit present a different behaviour. For this circuit and by using the cut sorting 4 and 5 (half of cuts threshold and unate, respectively), there are a significant improvement of around 30% in the logic depth with an increasing of less than 10% in the number of gates. Further investigation can be done since improving the logic depth can be crucial in this kind of arithmetic circuits.

5.2 Comparison to the state-of-the-art work

In order to compare the proposed synthesis flow to other approaches from different authors, three experiments were carried out. The first one compares the number of TLGs and the circuit logic depth to the results obtained from both strategies presented by Chen *et al.*, in (CHEN; WANG; CHANG, 2016), and also adopting a commercial tool. In the second, the proposed approach is compared to the Gowda's method, in (GOWDA et al., 2011), and the Palaniswami's method, in (PALANISWAMY; TRAGOUDAS, 2014), in terms of the number of TLGs. It is done because the most recent Chen's work, in (CHEN; WANG; CHANG, 2016), does not compare itself to those approaches. In the third experiment, the circuit area results is compared to the numbers presented in (ZHANG

et al., 2005) and in (LIN et al., 2014) in terms of the sum of input weights and function threshold value.

The comparison of our method to the approach presented by CHEN; WANG; CHANG, in (CHEN; WANG; CHANG, 2016), was carried out taking into account the IWLS 2005 benchmark suite (ALBRECHT, 2005). Table 5.5 shows the obtained results in terms of TLG count and circuit logic depth. Notice that the Chen's method already provides improvements of 28% in TLG count and 14% in logic depth when compared to the Zhang's method (ZHANG et al., 2005). Therefore, the Cheng's method has been adopted as reference.

When limiting the TLGs to 6 inputs, the proposed method reduced the TLG count in 94% of the circuits, with reductions up to 39% of such a count, being 20% on average. The logic depth is reduced in all applied benchmark circuits, with reductions up to 64% and 53% on average. The runtime is less than one second per circuit.

In order to exploit the gate level scalability, we have synthesized circuits for TLGs with up to 15 inputs. In this case, the TLG count was reduced in all circuits, with reductions up to 47% and 25% on average. The reduction in terms of logic depth is up to 67%, being 57% on average. In the same experiment, we have also synthesized the circuits by adopting a commercial tool. To do that, we provided the tool with a cell

Table 5.5: Comparison to the results presented by Chen (CHEN; WANG; CHANG, 2016) et al., in and by a commercial tool.

	number of TLGs				*circuit logic depth*			
circuit	*Chen et. al, 2016*	*Commercial Tool*	*Proposed K=6*	*Proposed K=15*	*Chen et. al, 2016*	*Commercial Tool*	*Proposed K=6*	*Proposed K=15*
spi	1614	(0.78)	(0.74)	(0.71)	19	(1.21)	(0.53)	(0.47)
systemcaes	5333	(0.82)	(0.79)	(0.77)	33	(0.88)	(0.42)	(0.42)
steppermotor	83	(0.75)	(0.71)	(0.61)	7	(2.57)	(0.43)	(0.43)
tv80	3559	(0.91)	(0.81)	(0.76)	30	(1.40)	(0.47)	(0.37)
ac97_ctrl	6194	(1.01)	(0.95)	(0.91)	7	(1.43)	(0.57)	(0.43)
sasc	333	(1.04)	(0.92)	(0.89)	7	(1.14)	(0.43)	(0.43)
pci_conf_cyc	62	(0.89)	(0.68)	(0.68)	4	(1.50)	(0.50)	(0.50)
usb_funct	6842	(0.93)	(0.82)	(0.78)	19	(1.42)	(0.53)	(0.47)
mem_ctrl	4721	(0.77)	(0.76)	(0.71)	23	(1.30)	(0.39)	(0.35)
systemcdes	1377	(0.90)	(0.82)	(0.77)	19	(1.16)	(0.47)	(0.47)
i2c	482	(0.87)	(0.76)	(0.69)	11	(1.45)	(0.36)	(0.36)
pci_bridge32	10497	(0.91)	(0.89)	(0.87)	21	(1.95)	(0.38)	(0.33)
aes_core	10057	(0.97)	(0.89)	(0.78)	17	(1.29)	(0.53)	(0.53)
simple_spi	436	(1.01)	(0.85)	(0.82)	8	(1.38)	(0.50)	(0.38)
des_area	2011	(1.01)	(1.01)	(0.95)	20	(1.40)	(0.55)	(0.50)
wb_conmax	21956	(0.87)	(0.82)	(0.80)	13	(1.62)	(0.62)	(0.54)
pci_spoci_ctrl	399	(0.64)	(0.61)	(0.53)	12	(1.50)	(0.42)	(0.33)
usb_phy	221	(0.85)	(0.70)	(0.64)	7	(1.14)	(0.43)	(0.43)
geomean		(0.88)	(0.80)	(0.75)		(1.39)	(0.47)	(0.43)

library composed by all NPN threshold functions with up to 6 variables. Notice that, although the commercial tool improves the Cheng's results in terms of TLG count, our method has been able to improve these results even more. On average, the commercial tool improves 7% whereas the proposed approach improves 15%. Moreover, the TLG count and circuit logic depth are simultaneously reduced by the proposed flow, whereas the commercial tool increases the Cheng's results in around 38% in terms of circuit logic depth.

On the other hand, in (PALANISWAMY; TRAGOUDAS, 2014), PALANISWAMY; TRAGOUDAS present two different improvements, called as BDD decomposition method (BDM) and ZDD decomposition method (ZDM), to the max literal factor tree (MLFT) method proposed by GOWDA et al., in (GOWDA et al., 2011). The results shown in Fig. 5.1 present the TLG count obtained by these methods and the one proposed herein. The ISCAS'85 set of benchmarks has been adopted for this evaluation. When compared to the MLFT approach, BDM and ZDM methods provide an average TLG count reduction of 12% and 17%, respectively. The average reduction obtained by our method is about 65% and 48% when compared to MLFT and ZDM, respectively.

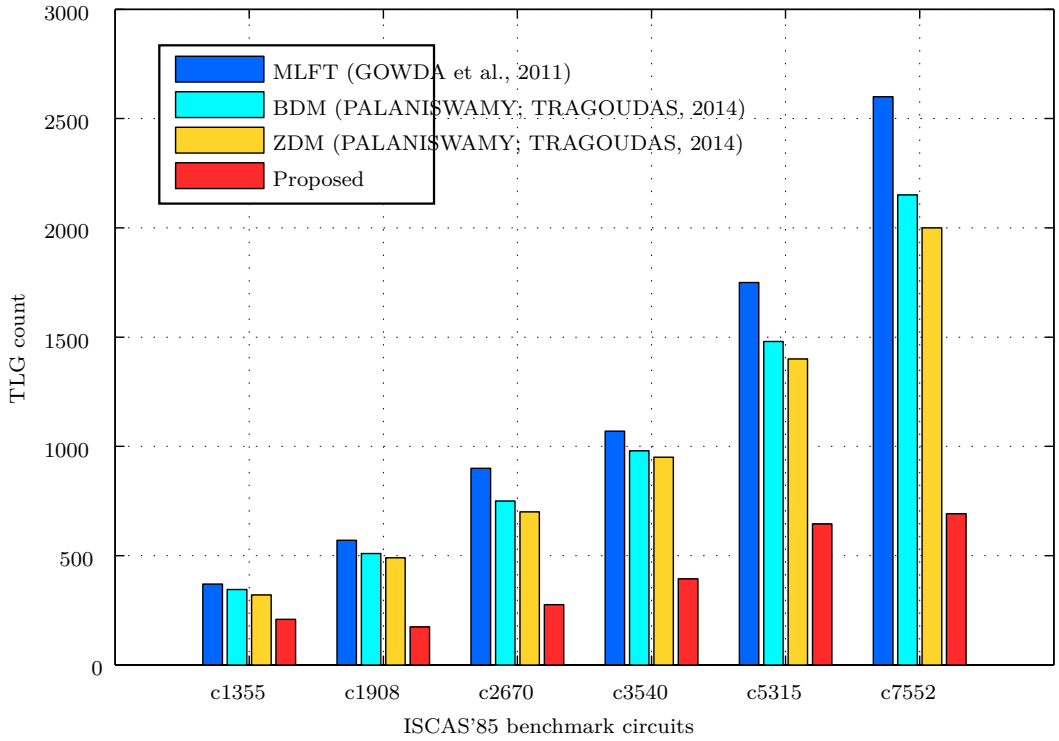


Figure 5.1: Comparing to the results presented by Palaniswamy (PALANISWAMY; TRAGOUDAS, 2014) and Gowda (GOWDA et al., 2011)

Finally, we compare our results to the work presented by LIN et al., in (LIN et al., 2014), in terms of the summation of input weights and threshold value. This method starts from a TLG netlist generated by the Zhang's method, in (ZHANG et al., 2005), and performs a rewiring procedure, optimizing the TLG area cost function. Table 5.6 shows the results from this experiment. In (LIN et al., 2014), Lin's method improves the Zhang's approach results for all benchmarks, obtaining a reduction of 4% on average. Our approach does not depend on a preliminary threshold synthesis and optimizes the cost function performing a threshold logic technology mapping directly over the original circuit. Therefore, we have improved the Zhang's results for all benchmarks, so reducing the circuit area in up to 46%, being 31% on average.

Table 5.6: Comparison to the area results from Zhang's (ZHANG et al., 2005) and Lin's (LIN et al., 2014) approaches.

	area estimation ($\sum W + T$) (ratio)				
circuit	*Zhang (ZHANG et al., 2005)*	*Lin (LIN et al., 2014)*		*Proposed*	
alu4	1986	1934	(0.97)	1973	(0.99)
apex6	2079	2007	(0.97)	1720	(0.83)
C1355	2102	2098	(1.00)	1312	(0.62)
C1908	1671	1631	(0.98)	1157	(0.69)
C5315	4661	4651	(1.00)	4133	(0.89)
C6288	9892	9844	(1.00)	7147	(0.72)
C7552	6468	6412	(0.99)	3972	(0.61)
dalu	3644	3608	(0.99)	2456	(0.67)
frg2	3299	2977	(0.90)	1928	(0.58)
i10	7490	6888	(0.92)	5472	(0.73)
i2c	3268	2867	(0.88)	2043	(0.63)
pair	4057	3945	(0.97)	3557	(0.88)
pci_spoci_ctrl	3254	3127	(0.96)	1472	(0.45)
rot	1960	1878	(0.96)	1550	(0.79)
s13207	9542	9221	(0.97)	5438	(0.57)
s9234	7056	6415	(0.91)	4149	(0.59)
simple_spi	2626	2540	(0.97)	2085	(0.79)
spi	12004	11184	(0.93)	6523	(0.54)
systemcdes	11677	11139	(0.95)	7190	(0.62)
usb_phy	1586	1498	(0.94)	1176	(0.74)
x3	2170	2054	(0.95)	1730	(0.80)
Geomean			(0.96)		(0.69)

5.3 Scalability analysis

In the last set of experiments, we evaluate the behaviour of the proposed approach when increasing the two main variables which the method depends: (*i*) the maximum number of inputs for each cut (K), and (*ii*) the maximum number of cuts stored for each AIG node (C). Figure 5.2 demonstrates two different graphs. The first graph presents the execution time variation, when increasing K and the second one presents the impacts in the resulting mapped circuit both in terms of the number of TLGs and the circuit logic depth. The value of K is increased from $K = 4$ to $K = 15$

The results show that the execution time increases more than 20 times when comparing $K = 15$ to $K = 4$, whereas the number of TLGs decreases around 24% and the logic depth decreases around 12%. It is also possible to notice that although the execution time is always increasing, the improvement in the mapped circuit is more significant up to 12 inputs. Moreover, we notice that the execution time behaviour seems to be linear up to 12 inputs (according to the expected), but seems to be quadratic for larger number of inputs. Through some experiments, we could conclude that for more than 12 inputs, the threshold identification time becomes predominant, and the method is quadratic in the number of inputs.

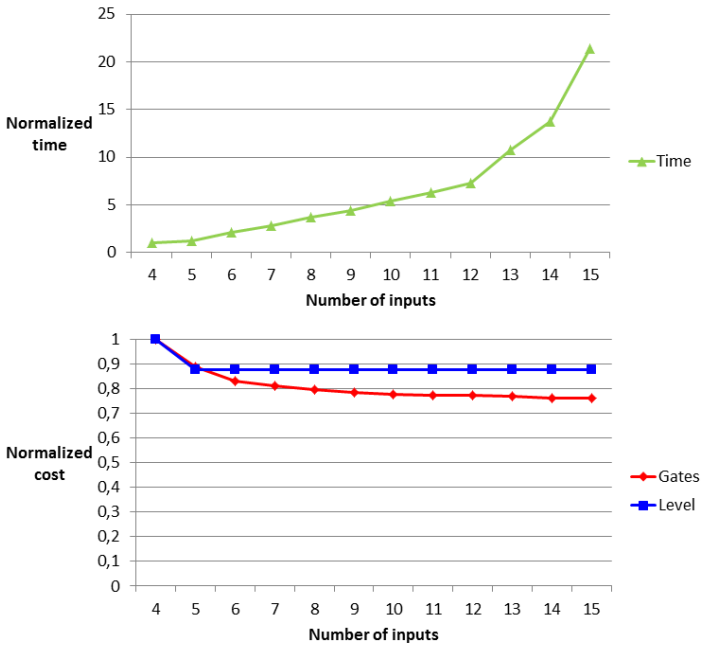


Figure 5.2: The behaviour of the proposed approach when increasing the maximum number of inputs for each cut (K).

Figu. 5.3 shows the results when increasing the number of cuts (C) stored for each AIG node. Three curves are presented, presenting the execution time, the number of TLGs and the logic depth in the mapped circuit. The C parameter is increased 8 by 8, from $C = 8$ to $C = 80$.

The results show that, although the execution time is always increasing, the quality of the mapped results is almost the same. The logic depth is not improved and the number of TLGs present a 1% of difference. This behaviour was the expected and is according to the priority cuts proposal (MISHCHENKO et al., 2007). Moreover, we notice that the execution time behaviour seems to be linear, although we expected a quadratic behaviour. We investigate and could conclude that, although the maximum number of cuts which can be stored for each AIG node increases, the real number of cuts stored is not always the maximum. The number of different cuts enumerated for each node is limited.

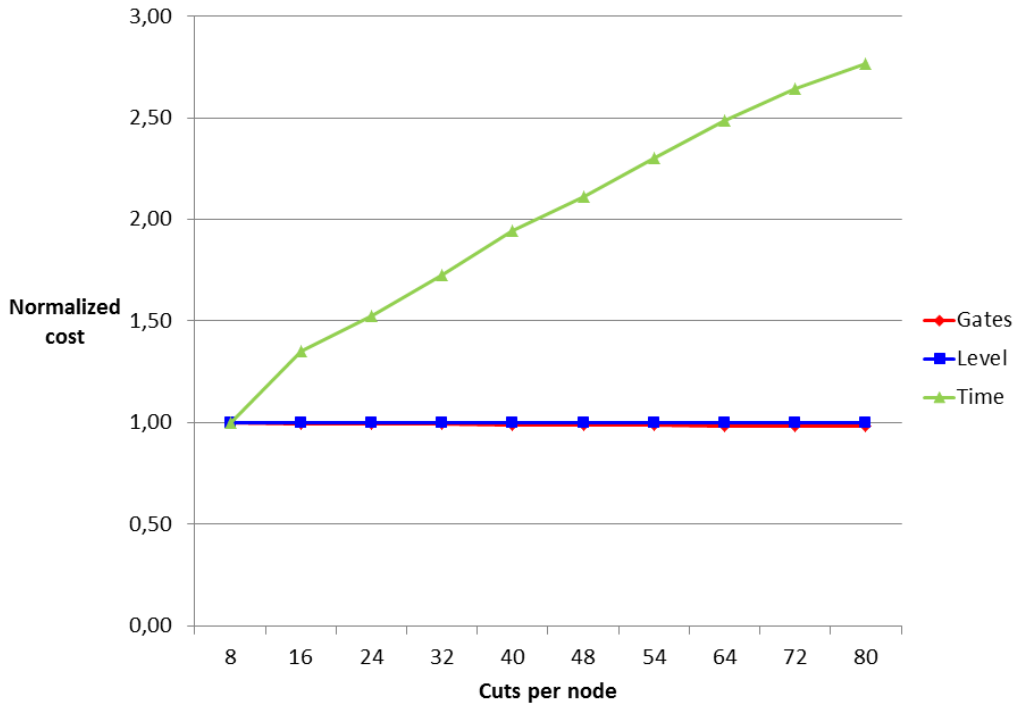


Figure 5.3: The behaviour of the proposed approach when increasing the maximum number of cuts stored for each AIG node (C).

Table 5.7: Summary of the obtained results for all the experimented cost functions and area estimations.

	area = #TLGs						area = $\sum$W + T						area = $\sum$#Inputs					
	area oriented		level oriented		relaxed level		area oriented		level oriented		relaxed level		area oriented		level oriented		relaxed level	
circuit	area	level	area	level	area	level	area	level	area	level	area	level	area	level	area	level	area	level
C1355	252	13	211	7	208	9	1312	13	2091	7	1648	9	656	13	908	8	868	10
C1908	175	18	192	10	174	13	1162	21	1637	10	1532	13	570	21	683	11	610	14
C2670	271	14	281	7	276	10	1777	13	2231	7	1955	9	828	15	869	10	835	13
C3540	396	24	398	13	394	16	2244	30	3155	13	2922	16	1361	25	1562	13	1396	18
C5315	650	18	669	10	645	11	4106	19	5648	9	4830	13	1993	18	2162	11	2052	14
C7552	678	32	714	10	692	11	3991	36	5645	10	4769	13	2150	30	2326	11	2240	14
oc_aquarius	8990	84	9179	29	9095	37	64776	92	83105	29	78035	37	28372	89	30406	32	29970	44
oc_cfft_1024x12	3943	22	3963	8	3961	10	22949	25	32365	8	30096	10	12826	26	14500	9	14170	11
oc_cordic_p2r	3787	20	3936	8	3886	9	27761	21	34454	8	32378	10	12370	22	13829	9	13331	11
oc_cordic_r2p	4720	22	4841	8	4794	9	32236	25	43353	8	41821	10	15455	27	17433	8	16998	10
oc_des_perf	8405	12	9136	7	8521	9	58212	13	84235	7	65235	9	29378	12	35223	7	31361	9
oc_ethernet	3648	24	3689	7	3651	9	23349	33	26493	7	25266	9	11695	33	12246	9	12072	11
oc_fpu	7816	948	8173	282	7973	360	43445	1016	79646	388	70018	361	23374	997	27616	291	26706	384
oc_mem_ctrl	6521	16	6588	8	6548	10	34363	25	41943	8	40089	10	19684	21	20317	11	19907	14
oc_video_dct	14003	29	14659	13	14290	16	98118	34	157479	13	149972	16	45729	31	55449	14	52370	18
oc_video_jpeg	18128	30	18879	12	18632	15	123335	34	171704	12	158158	15	60407	33	67014	14	65289	18
radar20	28782	45	30346	14	29719	18	161436	47	258722	14	229448	18	86378	45	98597	17	96131	19
uoft_raytracer	59232	70	60626	23	59925	29	355222	88	522004	23	481115	29	191468	85	211060	25	206744	32
adder	286	102	341	33	322	42	1967	129	3212	33	3069	42	906	129	1112	33	1095	42
bar	1792	14	1663	5	1406	7	8960	14	17856	5	13306	7	4480	14	5310	8	4941	10
div	17374	2117	20001	568	16392	752	85465	2272	144716	502	129789	705	47575	2200	43028	585	60450	790
hyp	129857	7337	137089	3189	134277	4167	731970	8778	1026321	3208	919501	4170	362848	8660	410824	3284	395135	4279
log2	16236	218	15570	91	15908	118	94205	221	142949	91	128127	119	46908	222	57093	97	52602	128
max	954	82	1398	35	1051	61	7089	179	16106	36	13199	49	3040	98	3891	56	3207	70
multiplier	15458	170	15279	62	15303	80	88022	179	119674	62	99787	78	41265	176	47965	64	43516	84
sin	2687	108	2690	43	2564	57	12844	141	28328	42	25839	54	7911	129	10505	46	10187	65
sqrt	11232	2169	12748	879	12585	1160	52842	2306	131820	885	121237	1160	31598	2167	42689	625	41867	803
square	9898	118	10007	42	9911	54	56593	132	67512	42	59345	49	28680	123	30738	50	28818	66
sixteen	4377156	50	4393330	31	4377179	40	5464453	87	5183416	31	5425647	40	4459711	53	4470871	34	4459728	44
twenty	5479332	54	5500745	34	5479282	44	7458200	69	7420364	34	8262487	44	5675698	53	5678591	39	5675698	52
twentythree	6127154	52	6148970	38	6126980	49	8541690	108	7812179	38	8395822	49	6364909	51	6369649	42	6364909	51

6 ADDITIONAL CONTRIBUTIONS

In this thesis, we proposed a new threshold logic synthesis flow based on identify threshold logic functions during the technology mapping. We have demonstrated that this approach results in improvements in terms of the number of TLGs and in terms of the circuit logic depth, when comparing to all the best works previously presented in the literature. In this chapter we show two additional contributions.

In Subsection 6.1, we demonstrate how to explore the threshold logic synthesis flow in order to synthesize circuits based on majority functions, a special case of TLFs. We propose how to implement any TLF in a majority gate with unbounded number of inputs and how to determine the minimum majority gate able to implement a given TLF. The obtained results overcome the state-of-the-art methods for 3-input and 5-input majority synthesis. Moreover, we present a guide for physical designers which are developing majority gates with a larger number of inputs. The objective is to evaluate which is a good area trade-off between a *maj-n* and a *maj*-3 gate.

Previous approaches for threshold logic synthesis adopt integer linear programming (ILP) to perform the threshold logic identification. The approach proposed in this thesis requires a very fast TLF identification method in order to identify if each enumerated cut represents a TLF or not. For this, we propose an ISOP-based heuristic where the inequality system is simplified and the weights are quickly assigned without solving the complete system. This approach represents an improvement of another method previously proposed in my master degree (NEUTZLING et al., 2017). We demonstrate the main differences that allowed the obtained improvements. The results demonstrate an excellent trade-off. The method is able to correctly identify more than 99% of the functions up to 16 inputs with 8 times of speedup. For large functions with up to 92 inputs, the speedup is around 4 orders of magnitude. This threshold logic identification approach is presented in Subsection 6.2.

6.1 Synthesis for Majority Based Circuits

A TLF is completely represented by a compact vector $[w_1,w_2,\ldots,w_n;T]$, where $w_1,w_2,\ldots,w_n$ are the input weights and T is the function threshold value. For instance, the corresponding TLG of the given functions $f = x_1x_2x_3$ and $g = x_1 + x_2 + x_3$ are $[1,1,1;3]$ and $[1,1,1;1]$, respectively.

A k-input majority function (maj-k), where k is an odd integer, is a special case of a threshold logic function where each input weight is 1 and the threshold value is $(k+1)/2$. For instance, the maj-3 and maj-5 are equivalent to the TLFs $[1,1,1;2]$ and $[1,1,1,1,1;3]$, respectively.

The technology mapping flow we propose exploits a standard FPGA technology mapping. The two main steps during FPGA technology mapping are the cut enumeration and covering. The cut enumeration decomposes the circuits into cuts that can be transformed into LUT. In turn, the covering phase chooses which of such cuts are actually used in the final circuit implementation. In this sense, the threshold method guarantees that only cuts that represent threshold functions are chosen during the cover phase.

In order to exploit the proposed threshold flow, we need a method to convert a threshold function into a majority gate. In this sense, to convert means to determine what is the number of inputs of the majority gate, which inputs are constants (0 or 1) and which inputs are tied together in order to represent the TLF input weight.

Since MAJ gates with a large number of inputs can be impractical, we set a maximum size k on the size of the majority gate. Therefore, during the cover phase, only threshold functions that fit into a k-input majority gate are allowed.

Several approaches were proposed in the recent years for synthesizing circuits based on majority gates (WANG; WALUS; JULLIEN, 2003; ZHANG et al., 2004; WALUS et al., 2004; IMRE et al., 2006; WANG et al., 2015; AMARU; GAILLARDON; MICHELI, 2016). After presenting the proposed approach, we compare the obtained results with the state-of-the-art work on majority synthesis.

6.1.1 Equivalence of threshold logic and majority logic

In this section, we show that any TLF can be converted into an unbounded majority gate. The proposed conversion is based on: (i) tying majority inputs together in order to obtain the required weight for a variable; and (ii) tying majority inputs to logic constant

1, in order to obtain the required threshold value. Thus, given a TLF F_{th} with m inputs, we obtain a majority gate G_{maj} with n_{min} inputs that implements F_{th}. The value of n_{min} depends on both the total sum of weights and on the threshold value of the target function F_{th}. In the following, T_{th} and T_{maj-n} denote the threshold value of F_{th} and of G_{maj}, respectively, and $\Delta T = T_{maj-n} - T_{th}$.

In a majority gate, each input has weight equal to 1. Therefore, a variable x_i of F_{th}, with weight w_i, must be connected to w_i inputs of G_{maj} to obtain the target weight. For instance, in order to implement the TLF $f_1 = [2, 1, 1, 1; 3]$ into a majority gate, variable x_1 must be connected to 2 inputs of the majority gate. In turn, each one of the remaining variables are connected to a single input of G_{maj}. Therefore, the minimum number of inputs n_{min} in G_{maj} to implement the target function is 5.

In the previous example, the target TLF has the same threshold value of the used majority gate. However, consider the case of converting the 4-variable TLF $f_2 = [1, 1, 1, 1; 2]$ into a 5-input majority gate[1]. Since, in this case, $\Delta T = 1$, one input of the majority gate should be tied to logic 1. This can be seen as if T_{maj} is reduced by one. In general case, the number of inputs tied to logic 1 is equals to ΔT.

In the following example, consider the case of converting the threshold function $f_3 = [2, 1, 1, 1; 2]$ into a majority gate g_1. Since the sum of weights of f_3 is 5, we begin by trying to fit it into a $maj - 5$ gate. As the weight of x_1 is 2, we assign two inputs of the $maj - 5$ to x_1 and obtain the function $[2, 1, 1, 1; 3]$. Since $\Delta T = 1$, one input of g_1 should be tied to logic 1. However, all the inputs of g_1 are already assigned to a variable of f_3. Thus, f_3 cannot be implemented in a $maj - 5$. Consequently, we try to fit f_3 into a $maj - 7$. As before, 5 inputs are assigned to variables of f_3. Therefore, there are 2 free inputs of the $maj - 7$ that can be used. Since $\Delta T = 2$, both free inputs are set to logic 1 and the target function $[2, 1, 1, 1; 2]$ is obtained.

From the previous examples, we can conclude that both the sum of weights and the threshold value impose lower bounds on the number of inputs the majority gate must have. In the following, we use:

$$W = \sum_{i=1}^{m} x_i w_i, \tag{6.1}$$

$$n_0 = (2 * T_{th}) - 1. \tag{6.2}$$

[1]Notice that majority gates always have an odd number of inputs.

Then, the following relationship must hold:

$$n_{min} \geq MAX(W, n_0). \tag{6.3}$$

Moreover, if $\Delta T > 0$, but there are no free inputs that can be connected to logic 1, the size of the majority gate must be increased and $2.\Delta T$ additional inputs are required.

In the previous examples, we applied a trial and error approach to obtain a majority implementation for a threshold function. In the following, we describe a more efficient method to obtain the value of n_{min}, which is shown in Algorithm 6.1. Initially, the algorithm checks which of (6.1) and (6.2) corresponds to the lower bound for n_{min}. If $n_0 \geq W$, then n_{min} is bounded by T_{th} and is equal to n_0. Otherwise, n_{min} is bounded by the sum of weights W. In this case, extra inputs are needed to obtain the correct threshold. The number of extra inputs is $2\Delta T$ when W is odd, or $2\Delta T - 1$ when W is even. Notice that the final value for n_{min} is always an odd number, as expected.

The method to obtain a majority gate implementation for a threshold function consists of 4 steps, which are executed in the following order:

1. **Obtain n_{min}:** as shown in Algorithm 6.1.

2. **Set the input weights:** for a variable x_i with weight w_i, assign x_i to w_i inputs of the majority gate.

3. **Adjust the threshold values:** if $\Delta T > 0$, then ΔT inputs of the majority gate are tied to logic 1.

4. **Handle unused inputs:** if there are unused inputs at the majority gate, all such inputs are tied to logic 0.

Algorithm 6.1: Obtaining the size of majority gate.

Input: TLF represented by $[w_1,w_2,\ldots,w_n;T]$
Output: the number of inputs k of the majority gate

$n_0 = (2 * T_{th}) - 1$;
$W = \sum_{i=1}^{m} w_i$;
if $n_0 \geq W$ **then**
    $n_{min} = n_0$;
elseif W ***is*** *odd* **then**
    $\Delta T = \frac{W+1}{2} - T_{th}$;
    $n_{min} = W + (2 * \Delta T)$;
else
    $\Delta T = \frac{W+2}{2} - T_{th}$;
    $n_{min} = W - 1 + (2 * \Delta T)$;
return n_{min};

6.1.2 Experimental Results on MAJ synthesis

In this section, we perform two experiments to demonstrate the efficacy of the proposed method for majority logic synthesis. The first experiment compares the proposed approach with the state-of-the-art methods (WANG et al., 2015; AMARU; GAILLARDON; MICHELI, 2016). The second experiment present results for the logic synthesis using majority gates with a larger number of inputs.

The proposed method is implemented in the ABC tool (Berkeley Logic Synthesis and Verification Group, 2017), being available to be adopted as a majority logic synthesis tool. The results presented in this section are generated through the command "*&if -a -M m -L l*". The parameter "*-M m*" enables the majority logic synthesis limiting the use of majority gates with up to m-inputs. In turn, the parameter "*-L l*" defines the gate area dependence on the number of inputs. Such an influence will be discussed in the second experiment.

6.1.2.1 Comparison to the state-of-the-art approach

The first set of experiments compares the results obtained by the proposed method to the state-of-the-art approach for $maj-3$ synthesis, proposed by WANG et al. (WANG et al., 2015) and by AMARU; GAILLARDON; MICHELI (AMARU; GAILLARDON; MICHELI, 2016). In order to allow a direct comparison, we adopt the same benchmark suites from the references. Namely, IWLS 2005 benchmark for (WANG et al., 2015) and

circuits taken from OpenCores for (AMARU; GAILLARDON; MICHELI, 2016).

Moreover, we also explored the use of $maj-5$ which previous works are unable to consider. In this experiment, we assume that both gates have the same area, which is a valid assumption for the USE methodology (CAMPOS et al., 2016b).

In Table 6.1, columns 2 and 3 present respectively the number of $maj-3$ gates and the logic depth obtained by Wang's method. Columns 4 and 5 present the results obtained by the proposed method restricted to $maj-3$. Improvements with respect to the number of gates and circuit logic depth are up to 36% (16% on average) and up to 32% (13% on average), respectively. Columns 6 and 7 show the results for $maj-5$ synthesis. In this case, the proposed approach reduces in up to %55 the number of gates (%46 on average) and up to 53% the logic depth (%26 on average) when compared to (WANG et al., 2015).

The work presented by AMARU; GAILLARDON; MICHELI, in(AMARU; GAILLARDON; MICHELI, 2016), introduces the majority-inverter graph (MIG) data structure. Although MIGs do not originally target emerging technologies, a MIG can be directly translated into a network of 3-input majority gates. Table 6.2 shows a trade-off between both methods when only $maj-3$ are used. Our method yields a smaller number of gates at the cost of increased logic depth. In this sense, the main advantage of the proposed method is the capability of using larger gates. For instance, when considering $maj-5$, we improve the average number of gates in more than 40% while also improving the average logic depth in around 17%.

Table 6.1: Comparison to the results obtained by the Wang's approach in (WANG et al., 2015).

	Wang 2015		Proposed (Maj3)		Proposed (Maj5)	
	Gates	Level	Gates	Level	Gates	Level
alu2	329	18	0.98	1.44	0.62	1.22
c8	108	8	0.99	0.88	0.60	0.88
k2	1193	19	0.72	0.68	0.49	0.58
frg2	568	15	1.07	0.80	0.63	0.73
apex6	662	17	0.86	0.71	0.55	0.59
example2	241	9	0.92	1.00	0.63	0.89
vda	670	14	0.64	0.79	0.45	0.79
frg1	102	17	0.65	0.71	0.38	0.47
x1	253	11	0.88	0.82	0.53	0.55
ttt2	144	10	0.82	1.10	0.53	1.00
geomean			0.84	0.87	0.54	0.74

Table 6.2: Comparison to the results obtained by the Amaru's approach in (AMARU; GAILLARDON; MICHELI, 2016).

	Amarú 2016		Proposed (Maj3)		Proposed (Maj5)	
	Gates	Level	Gates	Level	Gates	Level
systemcdes	2453	19	0.91	1.16	0.66	0.74
spi	3337	19	0.78	1.37	0.49	0.84
mem_ctrl	7143	19	0.94	1.21	0.61	0.74
tv80	7397	30	0.82	1.30	0.52	0.77
systemcaes	9547	25	0.83	1.16	0.54	0.80
ac97_ctrl	10745	8	0.96	1.13	0.61	0.75
usb_funct	12995	19	0.90	1.32	0.56	1.00
aes_core	20947	18	0.88	1.28	0.63	0.78
DSP	40517	34	0.87	1.68	0.56	1.35
des_perf	67194	15	1.03	1.07	0.76	0.67
geomean			0.89	1.26	0.59	0.83

6.1.2.2 $maj-n$ logic synthesis

In the second set of experiments, we explore the use of larger majority gates. When considering larger majority gates, it is important to take into account that the gate area can increase with the number of inputs. Therefore, reducing the number of gates may not always reduce the circuit area. In this sense, this experiment has two main goals: *(i)* evaluate the use of larger majority gates for different area estimation functions; and *(ii)* demonstrate the efficacy of the proposed method when handling $maj-n$ gates.

Herein, there is not a specific target technology. Therefore, it is important to consider a general metric for area estimation that can be used in different scenarios. The gate area, normalized to the $maj-3$ area, as function of the number of inputs is expressed by the following equation:

$$gateArea = (\frac{n}{3})^{\alpha} \tag{6.4}$$

where n is the number of inputs in the gate and α is a parameter to model the impact of the number of inputs on gate area. When $\alpha = 0$, all gates have the same area. When α=1 has been adopted as the area is proportional to the number of inputs. Notice that the area of the $maj-3$ gate is the reference, being equal to 1 for any α.

In the experiment, we consider the same circuits from Table 6.1. In Figure 6.1, each curve represents the total number of instances of $maj-3$, $maj-5$, $maj-7$ and $maj-9$ gates in the mapped networks, when varying α from 0 to 1.4.

We begin by evaluating the utilization of the $maj-5$ gate. The optimum implementation of the $maj-5$ using $maj-3$ gates contains 4 $maj-3$ (AKERS, 1962). Hence, the area of the $maj-5$ could be as much as four times larger than the area of the

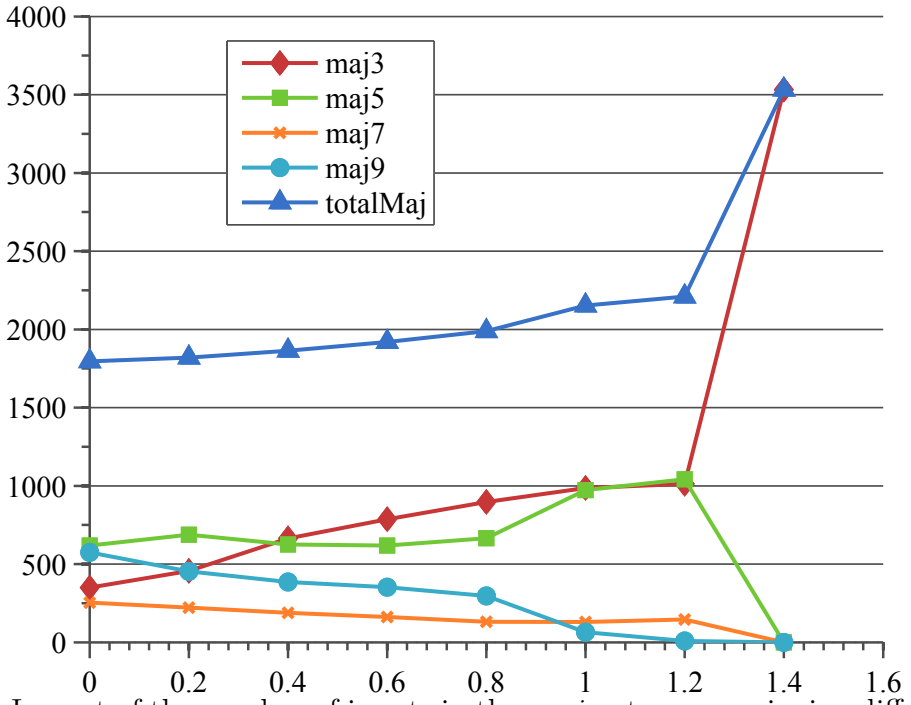


Figure 6.1: Impact of the number of inputs in the maj gate area assigning different values of the α parameter.

$maj-3$, leading to a maximum for α of 2.7. However, it is observed that no $maj-5$ has been used when $\alpha = 1.4$. The reason for this is that the most $maj-5$ gates has been used to implement OA21 ($f = x_0(x_1 + x_2)$) and AO21 ($f = x_0 + x_1x_2$) functions, which can be implemented using 2 $maj-3$ gates. In this case, the $maj-5$ area can be at most two times the $maj-3$ area, leading a maximum value for α of 1.36.

Some usual functions in circuits such as $f1 = x_3x_0(x_1+x_2)$ and $f2 = x_3+x_0+x_1x_2$ can be implemented in a $maj-9$ but not in a $maj-7$. This explains why the $maj-9$ is more used than $maj-7$ for $\alpha \leq 0.8$. In turn, for $\alpha \geq 0.8$, using a $maj-3$ and a $maj-7$ to implement each of $f1$ and $f2$ leads to a smaller area than using a $maj-7$.

Although this set of information is related to the evaluated benchmarks and to the technology mapping approach proposed in this work, it can be a useful guide to evaluate the trade-offs of gates with a larger number of inputs. Finally, we also performed the synthesis for majority gates with unbounded number of inputs and $\alpha = 0$. In this case, majority gates with up to 123 inputs have been used in the final circuit. This result indicates that majority gates with large number of inputs can be useful if the gate area is kept close to the $maj-3$ circuit area.

6.2 Threshold logic identification

The threshold logic identification is an essential task that corresponds to the process of identifying whether a given Boolean function is TLF, besides computing the variable weights and function threshold value. Notice that the most of the identification methods available in the literature is based on solving systems of inequalities generated from truth table description form. These methods exploit integer linear programming (ILP) to provide optimal results (ZHANG et al., 2005; AVEDILLO; QUINTANA, 2004; SUBIRATS; JEREZ; FRANCO, 2008). However, scalability is their main bottleneck because the system of inequalities tends to increase exponentially with the number of function variables.

One of the first heuristic (non-ILP based) methods to identify threshold functions was proposed by Gowda et al., in (GOWDA; VRUDHULA; KONJEVOD, 2007) and later improved in (GOWDA et al., 2011). Gowda's method applies functional decomposition and min-max factorization tree techniques. The target function is decomposed into simple sub-functions until these ones can be directly identified as AND and OR basic functions, or even as constant logic values '0' and '1'. These sub-functions are then merged by exploiting TLF properties.

In (TRAGOUDAS et al., 2010) Palaniswamy et al. proposed a method based directly on the Chow's parameters, being later improved in (PALANISWAMY; TRAGOUDAS, 2012). However, the main drawbacks of Palaniswamy's approach are the degradation on the number of identified TLFs and the fact that the assigned variable weights do not always correspond to the minimum possible values. These non-minimum weights may impact the final circuit area (ZHANG et al., 2005)(GAO; ALIBART; STRUKOV, 2013).

6.2.1 ILP-based approach

Equation 2.1 defines the relationship between the variable weights and the threshold value of the target TLF. If the function value is true ('1') for certain assignment vector then the sum of weights of this assignment is equal to or greater than the threshold value. Otherwise, the function value is false ('0'), *i.e.*, the sum of weights is less than the threshold value. From these relationships, it is possible to generate the associated inequalities. This relationship for the function $f_1 = x_1x_2 + x_1x_3x_4$ is illustrated in Table 6.3.

Table 6.3: Inequalities of the function $f_1 = x_1x_2 + x_1x_3x_4$ thruth table

x_1	x_2	x_3	x_4	f	Inequality
0	0	0	0	**0**	$0 < T$
0	0	0	1	**0**	$w_4 < T$
0	0	1	0	**0**	$w_3 < T$
0	0	1	1	**0**	$w_3 + w_4 < T$
0	1	0	0	**0**	$w_2 < T$
0	1	0	1	**0**	$w_2 + w_4 < T$
0	1	1	0	**0**	$w_2 + w_3 < T$
0	1	1	1	**0**	$w_2 + w_3 + w_4 < T$
1	0	0	0	**0**	$w_1 < T$
1	0	0	1	**0**	$w_1 + w_4 < T$
1	0	1	0	**0**	$w_1 + w_3 < T$
1	0	1	1	**1**	$w_1 + w_3 + w_4 \geq T$
1	1	0	0	**1**	$w_1 + w_2 \geq T$
1	1	0	1	**1**	$w_1 + w_2 + w_4 \geq T$
1	1	1	0	**1**	$w_1 + w_2 + w_3 \geq T$
1	1	1	1	**1**	$w_1 + w_2 + w_3 + w_4 \geq T$

Each sum of variable weights greater than the function threshold value is placed on the greater side set, whereas each sum of weights which is less than the threshold value belongs to the lesser side set. Each element on the greater side is greater than each element on the lesser side, and the greater side elements are greater than (or equal to) the threshold value, whereas the lesser side elements are smaller than that. The inequalities system is generated by performing a Cartesian product of the greater side set and the lesser side set.

The ILP-based methods than use linear programming to solve this generated inequalities system. If the system has a solution, the given Boolean function is threshold and the solution corresponds to the input weights and the function threshold value. Otherwise, the function is not a TLF.

6.2.2 TLF identification method proposed in my master thesis

During my master thesis, I proposed a heuristic method to address threshold logic identification (NEUTZLING et al., 2017). In such an approach, a complete system of inequalities is also built using a similar strategy to ILP inequalities generation algorithms. However, unlike ILP-based approaches, the inequalities system is not solved. Instead, the algorithm selects some of the inequalities as constraints to the associated variables to compute the variable weights in a bottom-up way. After this assignment, the consistency

of the complete system is verified in order to check whether the weights have been correctly computed. In the following, we briefly present the main steps of this previously proposed approach.

6.2.2.1 Generation of inequalities system from ISOP

In our method, redundancies in the initial inequalities system are avoided using two ISOP expressions, one for the direct function f and another for the negated function $!f$. In the example presented in Section 6.2.1, $f_1 = x_1x_2 + x_1x_3x_4$, the least true assignment vectors are (1,1,0,0) and (1,0,1,1), and the greatest false assignment vectors are (1,0,1,0), (1,0,0,1) and (0,1,1,1). Therefore, the algorithm creates only $(w_1 + w_2)$ and $(w_1 + w_3 + w_4)$ on the greater side, and $(w_1 + w_3)$, $(w_1 + w_4)$ and $(w_2 + w_3 + w_4)$ on the lesser side. Each sum of variable weights greater than the function threshold value is placed on the greater side set, whereas each sum of weights which is less than the threshold value belongs to the lesser side set.

6.2.2.2 Simplification of inequalities

After creating the inequalities system, the inequalities simplification process is performed through four basic tasks:

- Merging of variables with the same VWO parameter;
- Elimination of variables that appear in both sides of inequalities;
- Elimination of inequalities with no elements on the lesser side;
- Elimination of inequalities with a single element on the lesser side.

After the inequalities simplification and before computing the variable weights, the tuple <variables,inequalities> associating the variables with some of the inequalities is created, like the map in Figure 6.2. By making so, each variable points to inequalities in which the variable is present on the greater side. This relationship is exploited in the weight assignment step, discussed in the next section.

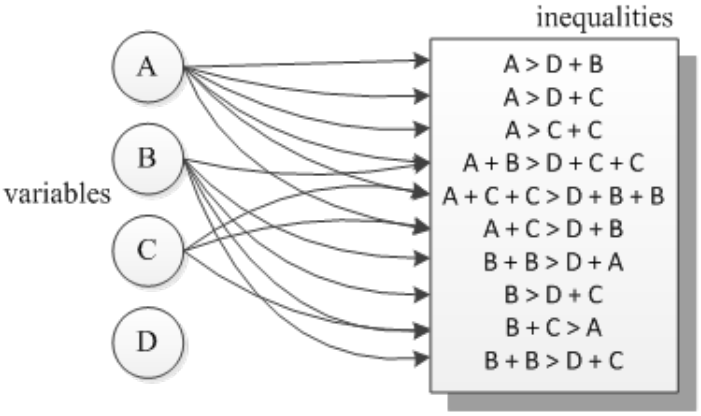


Figure 6.2: Example of relationship associating variables and inequalities .

6.2.2.3 Input weight assignment

The inputs weight assignment step receives the variables ordered by the Chow parameters, the inequalities and the map defined in the previous section. The first step is to assign minimum values for each variable. The variable with the lowest Chow parameter value is assigned by '1', the second smallest with '2', and so on. For each variable, in ascending ordering, the method checks the inequalities for which the variable points on the map (inequalities where such variable appears on the greater side). The checking is performed by simply verifying if the sum of the current values of the greater side variables is greater than the sum of the current values of the lesser side variables. If any of these inequalities is inconsistent, the value of the variable can be incremented, to try make it consistent.

In this previous version of the method, the input weight is incremented 1 by 1 and the inequality consistency is checked between each increment. The increments occurs either up to the inequality is consistent or up to the weight reach a defined limit. At the end, a single check is performed on the original system, replacing the variables with the values found. If all inequalities are consistent, the values are correct input weights.

After checking whether the input weights have been assigned correctly, the method calculates the function threshold value. In a TLF represented through an ISOP form, the sum of weights of the variables present in each product is equal to or greater than the threshold value. Therefore, the threshold value is equal to the least sum of weights of the greater side set.

6.2.3 Improvments in the TLF identification

In this thesis, we present a novel non-ILP based method to perform the identification and synthesis of TLF. The proposed approach presents three major contributions: (1) a new heuristic method to assign the variable weight values of threshold logic functions; (2) a novel ISOP-based procedure able to define the variable weight ordering before computing the absolute variable weight values; and (3) a fast algorithm to generate ISOP representation of unate Boolean functions.

(1) In the weight assignment of the previous approach, the algorithm creates a temporary variable, which controls the value assigned to the weights. Such a temporary value is initialized with a minimum value, being initially equal to 1, and is then increased one-by-one during the iterations. In the new method, each input weight is assigned in the first interaction, keeping the pre-computed order. When necessary, these weights are increased. In order to avoid many weight increasing steps, a "delta computation" step is introduced. As a result, this new weight assignment procedure provides significant improvements in the execution time with no loss in the number of identified TLFs.

(2) When the old version of the method received a truth table as input, the Espresso Logic Minimizer tool (BRAYTON et al., 1982) was used to generate the ISOP, but the execution time of this step was not computed in the results. In practical applications, often the input is a Boolean functions. For this reason, a fast ISOP generation method, specific for unate functions , is presented in this new version. The execution time of the ISOP generation is significantly less than the TLF identification method (the difference is around two orders of magnitude). Therefore, the ISOP is generated without significant impact in the execution time of the complete identification process. The execution time includes now the ISOP generation, and they remain similar to the execution time presented in the prior version, meaning that the presented gains are real.

(3) The old version of the method computes the Chow's parameters to determine the variable weight order. This step was performed based on the truth tables and the complexity is (2^n), being n the number of input variables. In this new version, a new parameter VWO is proposed, being computed directly from the ISOP, and now the complexity of the method depends on the number of cubes, which is $O(\frac{n!}{(\lfloor n/2 \rfloor !.\lceil n/2 \rceil !)})$ for unate functions, which is strictly smaller than 2^n.

6.2.3.1 ISOP generation based on Hasse diagram

The method proposed for threshold logic identification works over an ISOP representation of a given Boolean function f and an ISOP representation of $!f$. The ISOP needs to be generated when either the input function is represented by a truth table (usually for functions with up to 16 variables) or an ISOP of $!f$ is not provided.

The conventional methods adopted in the ISOP generation usually apply the Espresso Logic Minimizer tool (BRAYTON et al., 1982) and the Minato-Morreale algorithm (MINATO, 1993), being suitable for general (unate and binate) Boolean functions. In this work, we developed a faster procedure created specifically to handle unate functions.

This novel ISOP generation algorithm is based on the Hasse diagram, where each vertex corresponds to a minterm, each level i contains cubes with i literals, and each cube covers all of its descendants in the diagram (BIRKHOFF, 1940). A breadth-first search (BFS) is performed, such that larger cubes, *i.e.*, cubes that cover more minterms, are visited before the smaller cubes. For instance, a complete Hasse diagram of a 4-input Boolean function is illustrated in Fig. 6.3.

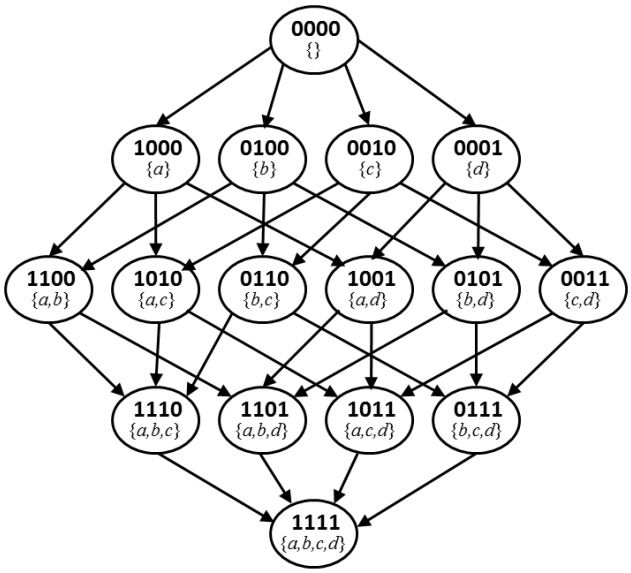


Figure 6.3: Hasse diagram of a 4-input Boolean function (BIRKHOFF, 1940).

For each visited vertex in the diagram, if the respective cube function f_c is contained in the target function f, then this cube is added to the partial solution f_p and its descendants are not visited. When the partial solution function is equivalent to the

target function ($f \equiv f_p$), the final solution is found. The ISOP generation procedure is described by the pseudo-code in Algorithm 6.2.

Algorithm 6.2: Compute the ISOP of a positive unate function based on the Hasse diagram.

```
   Input: Boolean function f
   Output: Set of cubes isop
 1 queue = ∅;
 2 isop=∅;
 3 status[ ]= ∅;
 4 f_p = ∅;
 5 foreach x_i ∈ X do
 6     queue.add(2^i);
 7 while queue is not empty do
 8     cube=queue.remove;
 9     if (status[parentof(cube)] = ACCEPTED) then
10         continue;
11     f_c = getfunction(cube);
12     if (f = f ∨ f_c) then
13         isop=isop ∪ cube;
14         status[cube]=ACCEPTED;
15         f_p = f_p ∨ f_c;
16     else
17         queue.add[childrenof(cube)];
18     if f_p = f then
19         return isop;
20 return isop;
```

6.2.3.2 Variable weight ordering computation

In the proposed method, the computation of the variable weight ordering is a crucial task because this information is used in the inequalities simplification and the weight assignment steps. A well-known way to obtain such an ordering is through the Chow's parameters (BRAYTON et al., 1982; MUROGA, 1971). The correlation between the Chow's parameters p_i and p_j of two variables x_i and x_j, respectively, induces the correlation between the corresponding weights w_i and w_j, *i.e.*, if $p_i > p_j$ then $w_i > w_j$ (WINDER, 1971).

A new algorithm to obtain a variable weight ordering parameter (VWO) is presented in this work. VWO parameters provide similar variable weight ordering as the Chow's ones, although the absolute parameter values are possibly different. These pa-

rameters are calculated directly from the ISOP, being quite straightforward and fast to compute. They are based on the max literal computation, as proposed in (GOWDA et al., 2011).

Considering an ISOP representation of a given function f, the largest variable weight of the target TLF is associated to the literal that occurs most frequently in the largest cubes of f, *i.e.*, in the cubes with fewer literals. In the case of a tie, it is decided by comparing the frequency of the literals in the next cubes with the smaller size.

The algorithm defines a weight for each cube, corresponding to the cube size. This weight is added to the VWO parameter of the variables present in the cube. The variable weight ordering is associated with the ordering of these parameters computed for each variable.

In the previous approach, the variable ordering is obtained through the Chow's parameters computation, whose time complexity is always 2^n for each variable. The complexity to compute the VWO parameter of each variable depends on the number of ISOP cubes, which is strictly smaller than 2^n. For unate functions, the worst case of the number of cubes is $(\frac{n!}{(\lfloor n/2 \rfloor !.\lceil n/2 \rceil !)})$.

6.2.3.3 New weights assignment approach

The variable weight assignment step receives the set of variables, ordered by the VWO parameters, as well as the inequalities and relationships defined in the final of Section 6.2.2.2. The first task is to assign minimum values for each variable. The variable with the lowest VWO parameter value is assigned by 1, the second lowest one by 2, and so on. This step is described in line 2 in Algorithm 6.3.

In the sequence, the algorithm iterates over all variables, in ascending order (according to the VWO ordering). Each variable points to a set of inequalities. The consistency of each inequality is verified, being performed by checking whether the sum of the current values of the greater side variables is greater than the sum of the current values of the lesser side variables.

The difference between the two inequality sides summation is called delta. If delta is not positive, the inequality is not consistent, *i.e.*, the current assigned weights do not satisfy this inequality. In this case, the value of the variable under verification is incremented, trying to make it valid. Delta is computed in line 6 in Algorithm 6.3, whereas the consistency is checked in line 7.

When the value of this variable is incremented, the value of the variables with

greater VWO parameter is also incremented in order to maintain the ordering. For instance, considering the following case $(A + C) > (B + B + B)$, increasing the value of C would never turn the inequality consistent because it also increases the values of A and B, i.e., $(A + 1 + C + 1) \geq (B + 1 + B + 1 + B + 1)$. In this sense, the lesser side cannot increase more than the greater side. The procedure *increment_weights* is responsible for incrementing the weight of variable v_i and the weight of the variables greater than v_i.

The decision whether the weight of a variable should be incremented is computed in three steps:

1. increment the variable value, as well as the value of the greater variables, by 1;

2. compute the new *delta*, denoted *delta*';

3. if $delta' \leq delta$, then the increment is undone and the inequality is kept as inconsistent, and the algorithm proceeds to the next inequalities and variables. When delta' > delta, the *increment_weights* procedure is applied.

The increment value that makes the inequality consistent is computed as follows:

$$increment = \lceil \frac{-delta}{delta - delta'} \rceil. \tag{6.5}$$

Finally, the original system is checked for consistency by replacing the variables by the assigned weights. If all inequalities are consistent, then the values correspond to the right variable weights. Otherwise, if at least one of the inequalities is not consistent then the method identifies the given function as not TLF. This ensures that false positive solutions are not found.

After checking whether the input weights have been assigned correctly, the method calculates the function threshold value. In a TLF represented through an ISOP form, the sum of weights of the variables present in each product is equal to or greater than the threshold value. Therefore, the threshold value is equal to the least sum of weights of the greater side set.

Algorithm 6.3: New and faster weight assignment approach.

```
   Input: : list_variables, relationship <variables , inequalities>
   Output: Array of weights W with assigned values
 1 foreach variable v_1 ∈ list_variables do
 2     w_i[v_i] = i
 3 foreach variable v_1 ∈ list_variables do
 4     ineq_set = get_ineqs_by_variable(v_i);
 5     foreach inequality ineq ∈ ineq_set do
 6         delta = weight_sum(greaterSide) − weight_sum(lesserSide);
 7         if delta ≤ 0 then
               /* this inequality is inconsistent                    */
 8             increment_weights(v_i, list_variables, 1);
 9             delta' = weight_sum(greaterSide) − weight_sum(lesserSide);
10             if delta' > delta then
11                 increment = −delta/(delta − delta');
12                 increment_weights(v_i, list_variables, increment);
13             else
14                 increment_is_undone;
```

6.2.4 Summary of improved results

The modifications has improved the method scalability. Although the main focus of the proposed method is to identify functions with at most 16 inputs, we now are able to identify larger functions (with up to 92 inputs). We would like to wrap-up by highlighting the contributions of this work compared to the related state-of-the-art approaches:

- Our method scales better and recognizes more threshold functions;
- Our method is the only one that compares numerically to ILP, whereas other authors only state that ILP is slow, without providing numerical results;
- We have reassessed the comparison with the ILP-based method. Our heuristic is around 8 times faster than the ILP, missing only 1% of threshold functions with up to 15 inputs present as sub-circuits in large opencore benchmark circuits;
- Our heuristic approach scales up to 92 inputs;
- For large functions, the proposed method is around 3 orders of magnitude faster than ILP.

7 FINAL REMARKS

This thesis presented an effective technology mapping for circuits using threshold logic gates, which overcome the state-of-the-art approaches. The main contributions of this work are the following:

- an efficient threshold logic synthesis flow based on cut pruning, which reduces area and delay, as well as is scalable to large benchmark circuits;
- a new priority cut strategy, where the cut sorting is based on the thresholdness of the Boolean functions implemented by each enumerated cut;
- a clever way to explore redundant cuts in order to improve the Quality-of-results;
- a technology mapping able to handle different TLG area estimations: the sum total of input weights and function threshold values, the total sum of gate inputs, and the total number of TLGs;
- a method for preforming majority gate synthesis, based on threshold logic functions.

When compared against the state-of-the-art related methods, the proposed approach reduces up to 47% the area and up to 67% the delay. We also present results customizing the developed mapper to achieve different mapping objectives (area x delay) and targeting different threshold-based area estimations, which can be used as reference in further publications. The proposed threshold logic synthesis is available through the public ABC tool.

Besides, the thesis presents improvements in a previous proposed threshold logic identification method, which is an enabler for the proposed technology mapping approach. Such an identification method is some orders of magnitude faster than the widely applied integer linear programming (ILP)-based strategy.

REFERENCES

AKERS, S. B. Synthesis of combinational logic using three-input majority gates. In: IEEE. SWITCHING CIRCUIT THEORY AND LOGICAL DESIGN. PROCEEDINGS OF THE THIRD ANNUAL SYMPOSIUM ON. **Proceedings...** [S.l.], 1962. p. 149–158.

ALBRECHT, C. Iwls 2005 benchmarks. In: INTERNATIONAL WORKSHOP FOR LOGIC SYNTHESIS (IWLS). **Proceedings...** [s.n.], 2005. p. 11–19. Available from Internet: <http://www.iwls.org>.

AMARU, L.; GAILLARDON, P.-E.; MICHELI, G. D. Majority-inverter graph: A new paradigm for logic optimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 35, n. 5, p. 806–819, 2016.

AVEDILLO, M. J.; QUINTANA, J. M. A threshold logic synthesis tool for rtd circuits. In: IEEE. DIGITAL SYSTEM DESIGN, 2004. DSD 2004. EUROMICRO SYMPOSIUM ON. **Proceedings...** [S.l.], 2004. p. 624–627.

BEIU, V.; QUINTANA, J.; AVEDILLO, M. VLSI implementations of threshold logic - A comprehensive survey. **IEEE Trans. on Neural Netw**, v. 14, n. 5, 2003.

Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. In: . [s.n.], 2017. Release 20170425. Available from Internet: <http://www.eecs.berkeley.edu/~alanmi/abc/>.

BERTACCO, V.; DAMIANI, M. The disjunctive decomposition of logic functions. In: IEEE COMPUTER SOCIETY. PROCEEDINGS OF THE 1997 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 1997. p. 78–82.

BIRKHOFF, G. **Lattice theory**. [S.l.]: American Mathematical Soc., 1940.

BRAYTON, R. K. et al. A comparison of logic minimization strategies using ESPRESSO: An APL program package for partitioned logic minimization. In: PROC. OF INT'L SYMP. ON CIRCUITS AND SYSTEMS (ISCAS). **Proceedings...** [S.l.: s.n.], 1982. p. 42–48.

CALLEGARO, V. et al. Bottom-up disjoint-support decomposition based on cofactor and boolean difference analysis. In: 2015 33RD IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN (ICCD). **Proceedings...** [S.l.: s.n.], 2015. p. 680–687.

CAMPOS, C. A. T. et al. Use: a universal, scalable, and efficient clocking scheme for qca. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 35, n. 3, p. 513–517, 2016.

CAMPOS, C. A. T. et al. Use: A universal, scalable, and efficient clocking scheme for qca. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 35, n. 3, p. 513–517, 2016.

CHEN, C.; MAO, Y. A statistical reliability model for single-electron threshold logic. **IEEE Transactions on Electron Devices**, IEEE, v. 55, n. 6, p. 1547–1553, 2008.

CHEN, D.; CONG, J. DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs. In: PROC. OF INT'L CONF. ON COMPUT.-AIDED DESIGN (ICCAD). **Proceedings...** [S.l.: s.n.], 2004.

CHEN, Y.-C.; WANG, R.; CHANG, Y.-P. Fast synthesis of threshold logic networks with optimization. In: IEEE. DESIGN AUTOMATION CONFERENCE (ASP-DAC), 2016 21ST ASIA AND SOUTH PACIFIC. **Proceedings...** [S.l.], 2016. p. 486–491.

CONG, J.; DING, Y. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. **IEEE Trans. on Comput.-Aided Design of Integr. Circuits Syst.**, v. 13, n. 1, 1994.

DERTOUZOS, M. L. Threshold logic a synthesis approach. 1965.

FAN, D.; SHARAD, M.; ROY, K. Design and synthesis of ultralow energy spin-memristor threshold logic. **IEEE Trans. on Nanotechnology**, v. 13, n. 3, 2014.

GAO, L.; ALIBART, F.; STRUKOV, D. B. Programmable cmos/memristor threshold logic. **IEEE Trans. on Nanotechnology**, v. 12, n. 2, 2013.

GOWDA, T.; VRUDHULA, S.; KONJEVOD, G. A non-ilp based threshold logic synthesis methodology. In: PROC. OF THE IWLS. **Proceedings...** [S.l.: s.n.], 2007. v. 8.

GOWDA, T. et al. Identification of threshold functions and synthesis of threshold networks. **IEEE Trans. on Comput.-Aided Design of Integr. Circuits Syst.**, v. 30, n. 5, 2011.

HINSBERGER, U.; KOLLA, R. Boolean matching for large libraries. In: ACM. PROCEEDINGS OF THE 35TH ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 1998. p. 206–211.

HU, S.-T. **Threshold logic**. [S.l.]: Univ of California Press, 1965.

IMRE, A. et al. Majority logic gate for magnetic quantum-dot cellular automata. **Science**, American Association for the Advancement of Science, v. 311, n. 5758, p. 205–208, 2006.

ITRS. **International Technology Roadmap for Semiconductors**. 2015.

KAHNG, A. B. The itrs design technology and system drivers roadmap: Process and status. In: ACM. PROCEEDINGS OF THE 50TH ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2013. p. 34.

KULKARNI, N.; VRUDHULA, S. Efficient enumeration of unidirectional cuts for technology mapping of boolean networks. **arXiv preprint arXiv:1603.07371**, 2016.

KULKARNI, N. et al. Reducing power, leakage, and area of standard-cell asics using threshold logic flip-flops. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 24, n. 9, p. 2873–2886, 2016.

KUO, P.-Y.; WANG, C.-Y.; HUANG, C.-Y. On rewiring and simplification for canonicity in threshold logic circuits. In: PROC. OF INT'L CONF. ON COMPUT.-AIDED DESIGN. **Proceedings...** [S.l.: s.n.], 2011.

LAGEWEG, C.; COTOFANA, S.; VASSILIADIS, S. A linear threshold gate implementation in single electron technology. In: IEEE. VLSI, 2001. PROCEEDINGS. IEEE COMPUTER SOCIETY WORKSHOP ON. **Proceedings...** [S.l.], 2001. p. 93–98.

LENT, C. S. et al. Quantum cellular automata. **Nanotechnology**, IOP Publishing, v. 4, n. 1, p. 49, 1993.

LEWIS, P. M.; COATES, C. L. Threshold logic. 1967.

LIN, C.-C. et al. Rewiring for threshold logic circuit minimization. In: PROC. OF CONF. ON DESIGN, AUTOMATION & TEST IN EUROPE. **Proceedings...** [S.l.: s.n.], 2014.

MAAN, A. K.; JAYADEVI, D. A.; JAMES, A. P. A survey of memristive threshold logic circuits. **IEEE transactions on neural networks and learning systems**, IEEE, v. 28, n. 8, p. 1734–1746, 2017.

MANOHARARAJAH, V.; BROWN, S. D.; VRANESIC, Z. G. Heuristics for area minimization in lut-based fpga technology mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 25, n. 11, p. 2331–2340, 2006.

MARTINS, M. G.; RIBAS, R. P.; REIS, A. I. Functional composition: A new paradigm for performing logic synthesis. In: IEEE. QUALITY ELECTRONIC DESIGN (ISQED), 2012 13TH INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2012. p. 236–242.

MARTINS, M. G. et al. Boolean factoring with multi-objective goals. In: IEEE. COMPUTER DESIGN (ICCD), 2010 IEEE INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.], 2010. p. 229–234.

MINATO, S.-i. Fast generation of prime-irredundant covers from binary decision diagrams. **IEICE transactions on fundamentals of electronics, communications and computer sciences**, The Institute of Electronics, Information and Communication Engineers, v. 76, n. 6, p. 967–973, 1993.

MISHCHENKO, A.; BRAYTON, R. Faster logic manipulation for large designs. In: PROC. OF INT'L WORKSHOP ON LOGIC AND SYNTHESIS. **Proceedings...** [s.n.], 2007. p. 1–6. Available from Internet: <http://people.eecs.berkeley.edu/~alanmi/publications/>.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. K. Improvements to technology mapping for LUT-based FPGAs. **IEEE Trans. on Comput.-Aided Design of Integr. Circuits Syst.**, v. 26, n. 2, 2007.

MISHCHENKO, A. et al. Combinational and sequential mapping with priority cuts. In: IEEE PRESS. PROCEEDINGS OF THE 2007 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 2007. p. 354–361.

MUROGA, S. **Threshold logic and its applications**. [S.l.: s.n.], 1971.

NAVI, K. et al. Five-input majority gate, a new device for quantum-dot cellular automata. **Journal of Computational and Theoretical Nanoscience**, American Scientific Publishers, v. 7, n. 8, p. 1546–1553, 2010.

NEUTZLING, A. et al. A Simple and Effective Heuristic Method for Threshold Logic Identification. **IEEE Trans. on Comput.-Aided Design of Integr. Circuits Syst.**, 2017. To be published.

NEUTZLING, A. et al. Synthesis of threshold logic gates to nanoelectronics. In: IEEE. INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI), 2013 26TH SYMPOSIUM ON. **Proceedings...** [S.l.], 2013. p. 1–6.

NEUTZLING, A. et al. A constructive approach for threshold logic circuit synthesis. In: IEEE. CIRCUITS AND SYSTEMS (ISCAS), 2014 IEEE INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2014. p. 385–388.

NEUTZLING, A. et al. Threshold logic synthesis based on cut pruning. In: IEEE PRESS. PROCEEDINGS OF THE IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 2015. p. 494–499.

NEUTZLING, A. et al. **Improvements to Technology Mapping for Threshold Logic Gates: Datasets and Technical Report**. 2017. Available from Internet: <http://dx.doi.org/XX.YYYYY/xrpfs4rs78h.1>.

NUKALA, N. S.; KULKARNI, N.; VRUDHULA, S. Spintronic threshold logic array (stla)—a compact, low leakage, non-volatile gate array architecture. **Journal of Parallel and Distributed Computing**, Elsevier, v. 74, n. 6, p. 2452–2460, 2014.

PACHA, C. et al. Threshold logic circuit design of parallel adders using resonant tunneling devices. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 8, n. 5, p. 558–572, 2000.

PALANISWAMY, A. K.; TRAGOUDAS, S. An efficient heuristic to identify threshold logic functions. **ACM Journal on Emerging Technologies in Computing Systems (JETC)**, ACM, v. 8, n. 3, p. 19, 2012.

PALANISWAMY, A. K.; TRAGOUDAS, S. Improved threshold logic synthesis using implicant-implicit algorithms. **ACM Journal on Emerg. Tech.**, v. 10, n. 3, 2014.

PAN, P.; LIN, C.-C. A new retiming-based technology mapping algorithm for lut-based fpgas. In: ACM. PROCEEDINGS OF THE 1998 ACM/SIGDA SIXTH INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS. **Proceedings...** [S.l.], 1998. p. 35–42.

PISTORIUS, J. et al. Benchmarking method and designs targeting logic synthesis for fpgas. In: PROC. OF INT'L WORKSHOP ON LOGIC AND SYNTHESIS. **Proceedings...** [S.l.: s.n.], 2007. v. 7.

SHENG, Q. **Threshold logic**. [S.l.]: Academic Press, 1969.

SUBIRATS, J. L.; JEREZ, J. M.; FRANCO, L. A new decomposition algorithm for threshold synthesis and generalization of Boolean functions. **IEEE Trans. on Circuits Syst. I**, v. 55, n. 10, 2008.

TRAGOUDAS, S. et al. Scalable identification of threshold logic functions. In: ACM. PROCEEDINGS OF THE 20TH SYMPOSIUM ON GREAT LAKES SYMPOSIUM ON VLSI. **Proceedings...** [S.l.], 2010. p. 269–274.

WALUS, K. et al. Circuit design based on majority gates for applications with quantum-dot cellular automata. In: IEEE. SIGNALS, SYSTEMS AND COMPUTERS, 2004. CONFERENCE RECORD OF THE THIRTY-EIGHTH ASILOMAR CONFERENCE ON. **Proceedings...** [S.l.], 2004. v. 2, p. 1354–1357.

WANG, P. et al. Synthesis of majority/minority logic networks. **IEEE Transactions on Nanotechnology**, IEEE, v. 14, n. 3, p. 473–483, 2015.

WANG, W.; WALUS, K.; JULLIEN, G. A. Quantum-dot cellular automata adders. In: IEEE. NANOTECHNOLOGY, 2003. IEEE-NANO 2003. 2003 THIRD IEEE CONFERENCE ON. **Proceedings...** [S.l.], 2003. v. 1, p. 461–464.

WINDER, R. O. Chow parameters in threshold logic. **Journal of the ACM (JACM)**, ACM, v. 18, n. 2, p. 265–289, 1971.

ZHANG, R. et al. Threshold network synthesis and optimization and its application to nanotechnologies. **IEEE Trans. on Comput.-Aided Design of Integr. Circuits Syst.**, v. 24, n. 1, 2005.

ZHANG, R. et al. A method of majority logic reduction for quantum cellular automata. **IEEE Transactions on Nanotechnology**, IEEE, v. 3, n. 4, p. 443–450, 2004.