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

OSVALDO MARTINELLO JUNIOR

KL-Cuts: A New Approach for Logic Synthesis Targeting Multiple Output Blocks

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

Prof. Dr. Renato Perez Ribas
Advisor

Prof. Dr. André Inácio Reis
Co-advisor

Porto Alegre, October 2010

CIP – CATALOGING-IN-PUBLICATION

Martinello Junior, Osvaldo

KL-Cuts: A New Approach for Logic Synthesis Targeting Multiple Output Blocks / Osvaldo Martinello Junior. – Porto Alegre: PPGC da UFRGS, 2010.

85f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Advisor: Renato Perez Ribas; Co-advisor: André Inácio Reis.

1. AIG. 2. Cut Enumeration. 3. KL-Cuts. 4. Logic Design. 5. Logic Synthesis. 6. Multiple Output Blocks. 7. Technology Mapping. I. Ribas, Renato Perez. II. Reis, André Inácio. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"This is your life and it's ending one minute at a time."

— CHUCK PALAHNIUK (FIGHT CLUB)

ACKNOWLEDGMENTS

It is a pleasure to thank the many people who made this thesis possible.

I am grateful to my advisor, Renato Perez Ribas, for his time and for his sense of organization (which I lack). His understanding, encouraging and personal guidance have provided a good basis for the present thesis.

I would like also to express my gratitude to my co-advisor, André Inácio Reis. His wide knowledge and his logical way of thinking have been of great value for me.

I could not forget to thank my labmates, for supporting my work, for the stimulating discussions, for working together before deadlines, and for all the fun we have had in the last years.

To my oldest friends from Mato Grosso whose friendship molded me as I am, and to my friends who are next to me now with whom I have divided the latest cheer times, my sincere thank you.

I am especially thankful to my family: to my father Osvaldo Martinello, to my mother Cleci Maria Martinello and to my sisters Christine and Caroline. I have missed them a lot during this research time.

I owe my deepest gratitude to my girlfriend Bibiana Strohmayer Alves. Without her encouragement, understanding and love it would have been impossible for me to finish this work.

I need also to thank the ones that funded this period of research (besides my father). The company Nangate Inc under a Nangate/UFRGS research agreement, CNPq Brazilian funding agency through the National Program of Microelectronics (PNM), and the European Community's Seventh Framework Programme under grant 248538 - Synaptic. Without their investment this work would not be possible.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 9

LIST OF FIGURES 11

LIST OF TABLES 13

ABSTRACT 15

RESUMO 17

1 INTRODUCTION 19
1.1 Synthesis 19
1.2 Motivation 20
1.3 Objective 20
1.4 Thesis Organization 21

2 TECHNICAL BACKGROUND 23
2.1 Boolean Function and Boolean Network 23
2.2 Equivalence Classes of Logic Functions 24
2.3 Data Structures 25
2.3.1 Directed Acyclic Graph 26
2.3.2 Forest of Trees 26
2.3.3 And-Inverter Graph 26
2.3.4 Binary Decision Diagram 27
2.4 Dag Nodes and Tree Nodes 28
2.5 K-Feasible Cuts 29
2.6 Library 30
2.7 Technology Mapping 30
2.7.1 Decomposition 30
2.7.2 Pattern Matching 31
2.7.3 Covering 31

3 STATE OF THE ART 33
3.1 Technology Mapping 33
3.2 DAG-Aware AIG rewriting 36
3.3 Using Signatures on Cut Computation 37
3.4 Factor Cuts 38
3.4.1 Complete Cut Factorization 38
3.4.2 Partial Cut Factorization 39

3.5 TEMPLATE Boolean Matching Method 39
3.6 Area Flow Covering 42

4 KL-FEASIBLE CUTS 45
4.1 L-Feasible Backcuts 45
4.1.1 Factor Backcuts 46
4.2 KL-Cuts Generation Algorithm 47
4.3 Unbounded KL-Cuts 49
4.3.1 KL-Cuts with unbounded K 50
4.3.2 KL-Cuts with unbounded L 51

5 APPLICATIONS OF KL-CUTS 53
5.1 Technology Mapping 53
5.1.1 Greedy Covering 53
5.1.2 Area Flow Covering for Multiple Outputs 54
5.1.3 Matching 58
5.1.4 Partitioning 61
5.2 Regularity Extraction 61
5.3 Peephole Optimization 62

6 RESULTS 63
6.1 L-Feasible Backcuts 63
6.2 KL-Cuts 64
6.3 Covering Algoritms 65
6.3.1 Greedy Covering 67
6.3.2 Area Flow Covering for Multiple Output 69
6.3.3 Multiple Output Matching 70
6.3.4 Partitioning 70

7 CONCLUSIONS AND FUTURE WORK 73

REFERENCES 75

A APPENDIX <KL-CUTS: UMA NOVA ABORDAGEM PARA SíNTESE LÓGICA UTILIZANDO BLOCOS COM MÚLTIPLAS SAíDAS> 79
A.1 Introdução 79
A.2 Cortes-KL 80
A.2.1 Aplicações 80
A.3 Conclusões 84

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-Inverter Graph
BDD	Binary Decision Diagram
CAD	Computer-Aided Design
DAG	Directed Acyclic Graph
DFM	Design for Manufacturing
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
IPO	In-Place Optimization
LUT	Lookup Table
MFFC	Maximum Fanout-Free Cone
OTR	Odd-level Transistor Replacement
PI	Primary Input
PO	Primary Output
RTL	Register Transfer Level
TSBDD	Terminal-Suppressed Binary Decision Diagram
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

LIST OF FIGURES

Figure 2.1: A truth table representation of a function. 24
Figure 2.2: A Boolean network. 24
Figure 2.3: A directed acyclic graph example. 26
Figure 2.4: A forest of trees. 27
Figure 2.5: An and-inverter graph. 27
Figure 2.6: An example of a BDD. 28
Figure 2.7: AIG illustrating dag and tree nodes. 28

Figure 3.1: Different AIG structures for function $f = a * b * c$. 36
Figure 3.2: Two cases of AIG rewriting of a node. 36
Figure 3.3: Naive approach for computing $R[f]_P$. 40
Figure 3.4: A generic view of a truth table. 40
Figure 3.5: Reducing search space by cutting non-maximal branches. 41
Figure 3.6: Reducing search space by using symmetry. 42

Figure 4.1: AIG demonstrating backcut factorization. 46
Figure 4.2: Pseudo-code for *KL*-cuts calculation. 48
Figure 4.3: AIG illustrating covering. 49
Figure 4.4: Pseudo-code for *KL*-cuts with unbounded K computation. 50
Figure 4.5: AIG exemplifying *KL*-cuts with unbounded K. 51
Figure 4.6: Pseudo-code for *KL*-cuts with unbounded L computation. 52

Figure 5.1: An AIG to illustrate the multiple output area flow algorithm. 57
Figure 5.2: An AIG with a loop formed by *KL*-cuts. 58
Figure 5.3: A truth table representation of a set of functions L_1. 60
Figure 5.4: Computing $R[L_1]_{PP}$. 61

Figure 6.1: Number of *KL*-cuts versus K. 65
Figure 6.2: Time taken to compute *KL*-cuts versus K. 65
Figure 6.3: Number of *KL*-cuts versus L. 66
Figure 6.4: Time taken to compute *KL*-cuts versus L. 66
Figure 6.5: Execution time of matching algorithm varying the number of inputs. . 71
Figure 6.6: Execution time of matching algorithm varying the number of outputs. 71

Figure A.1: Tempo de execução de algoritmo de identificação de padrões variando o número de entradas. 84
Figure A.2: Tempo de execução de algoritmo de identificação de padrões variando o número de saídas. 85

LIST OF TABLES

Table 2.1: Number of equivalence classes under various equivalence relations. . 25
Table 2.2: An example of *K*-feasible cuts computation. 29

Table 3.1: Effect of iterations in area flow. 43

Table 4.1: An example of *L*-feasible backcuts computation. 47

Table 5.1: Effect of iterations in multiple output area flow, using only mode 1. . 56
Table 5.2: Effect of iterations in multiple output area flow, using all modes of operation. 56
Table 5.3: Area flow according to different modes of computing fanout. 56

Table 6.1: Benchmark information. 63
Table 6.2: Comparison between *L*-backcut enumeration and factor *L*-backcut enumeration. 64
Table 6.3: Comparison between *KL*-cuts enumeration and factor *KL*-cuts enumeration. 67
Table 6.4: Covering for single output LUTs using ABC and Area Flow methods. 68
Table 6.5: Greedy bottom-up covering using *KL*-cuts. 68
Table 6.6: Greedy bottom-up covering using only factor *KL*-cuts. 69
Table 6.7: Covering using the area flow for multiple outputs algorithm. 70
Table 6.8: Comparison between a covering with factor trees and a covering with *KL*-cuts with unbounded K. 72

Table A.1: Cobertura para LUTs de uma saída usando ABC e fluxo de área. . . . 81
Table A.2: Cobertura gulosa usando cortes-*KL*. 82
Table A.3: Mapeamento de fluxo de área para múltiplas saídas. 83

ABSTRACT

This thesis introduces the concept of *KL*-feasible cuts, which allows controlling both the number K of inputs and the number L of outputs in a circuit region. The design of a digital circuit can roughly be divided in two phases: logic synthesis and physical synthesis. Within logic synthesis, one of the main steps is the technology mapping. Traditionally, the technology mapping process only handles single output functions, in order to construct circuits. The objective of this method is to explore the use of multiple output blocks on technology mapping. To provide scalability, the concept of factor cuts is extended to *KL*-cuts. Algorithms for enumerating these cuts and also for enumerating some subsets of cuts with some special characteristics are presented and results are shown. As examples of practical applications, different covering algorithms are proposed. The greedy algorithm is a simple alternative and produces good results in area, but it is too restrictive, as it is not practical in timing oriented mapping. The other covering algorithm presented is an extension to the area flow algorithm and allows cuts with multiple outputs to be used while making possible the control of some other costs. A Boolean matching algorithm that is able to handle multiple output blocks is also described, which permits the use of a standard cell library with more than one output on technology mapping. The results show the viability and usefulness of the method.

Keywords: AIG, Cut Enumeration, KL-Cuts, Logic Design, Logic Synthesis, Multiple Output Blocks, Technology Mapping.

KL-Cuts: Uma Nova Abordagem para Síntese Lógica Utilizando Blocos com Múltiplas Saídas

RESUMO

Esta dissertação introduz o conceito de cortes KL, o que permite controlar tanto o número K de entradas como o número L de saídas em uma região de um circuito. O projeto de um circuito digital pode ser dividido em duas fases: síntese lógica e síntese física. Dentro de síntese lógica, um dos principais passos é o mapeamento tecnológico. Tradicionalmente, o processo de mapeamento tecnológico somente lida com funções de saída única, para a construção de circuitos. O objetivo deste método é explorar o uso de blocos de múltiplas saídas no mapeamento tecnológico. Para prover escalabilidade, o conceito de fatoração de cortes é estendido para os cortes KL. Algoritmos para enumerar esses cortes e também para enumerar alguns subconjuntos de cortes com características específicas são apresentados e os resultados são mostrados. Como exemplos de aplicações práticas, diferentes algoritmos de cobertura são propostos. O algoritmo guloso é uma alternativa simples e produz bons resultados em área, mas é muito restritivo, pois não é factível em mapeamento orientado à atraso. Outro algoritmo de cobertura apresentado é uma extensão do algoritmo de fluxo de área e permite a utilização de cortes com várias saídas, mantendo possível a consideração de outros custos. Um algoritmo de correspondência Booleana que é capaz de lidar com blocos com múltiplas saídas também é descrito. Isso permite a utilização de uma biblioteca padrão com células com mais de uma saída no mapeamento tecnológico. Os resultados mostram a viabilidade e utilidade do método.

Palavras-chave: AIG, Blocos com Múltiplas Saídas, Enumeração de Cortes, KL-Cuts, Mapeamento Tecnológico, Projeto Lógico, Síntese Lógica.

1 INTRODUCTION

Technologies based on digital integrated circuits have major impact on society, being present on virtually every knowledge area. The advances in the field of conception of integrated circuits make possible the aggregation of an increasingly large number of components on a same device. This high integration scale imposes new challenges to the synthesis process. In order to deal with constant changes in the design rules, and to increase productivity, the automation of this process through the use of EDA (Electronic Design Automation) tools plays a crucial role.

Usually the design methodologies are classified as custom and semicustom design (MICHELI, 1994). In the former methodology, both functional and physical designs are handcrafted, requiring hard skilled designers and a great effort in order to fine-tune features of the circuit. This methodology has, therefore, a high cost, which may be compensated by a high quality design. Semicustom design consists in establishing design restrictions, such as limiting the number of primitives, which limit the ability of optimization of a circuit. This reduction on the solution space makes easier the development of CAD (Computer-Aided Design) tools for design and optimizations, reducing the time-to-market of a design. Currently the number of semicustom designs outnumbers custom design.

1.1 Synthesis

The goal of the circuit synthesis is to transform a higher abstraction level description of a circuit into a more detailed model, such as a geometrical model. The whole synthesis process is often broken into three major steps: architectural synthesis, logic synthesis and physical synthesis (MICHELI, 1994). The architectural synthesis, often called high-level synthesis, consists of transforming a behavioral description of a system — usually represented in an HDL (Hardware Description Language), such as VHDL or Verilog, or even in a higher abstraction level language, such as SystemC — into a structural view, which describes the organization of the system, mainly described in RTL (Register Transfer Level).

The next step is the logic synthesis, whose role is to translate a logic description of a circuit into a network of interconnected cells of a given technology. It is typically divided in three phases: technology independent optimizations, technology mapping and technology dependent optimizations. The first one applies some transformations that do not depend on the technology, but depend on the chosen mapping algorithm. These transformations can be structural or Boolean. Then the technology mapping phase binds the circuit with the technology, by mapping portions of the circuit to a cell implemented in the target technology. After that, more optimizations are applied to the mapped circuit, such

as cell resizing or logic duplication. These are called technology dependent optimizations.

The physical synthesis, or geometrical level synthesis, physically distributes the cells and performs its interconnections. The final product is a layout of the circuit that implements the initial behavioral description of the system.

1.2 Motivation

Some recent advances on logic synthesis are based on And-Inverter Graphs (AIGs), for scalability reasons (LING; ZHU; BROWN, 2008; MISHCHENKO; BRAYTON, 2006). Part of these advances is based on the concept of *K*-feasible cuts (CONG; WU; DING, 1999; PAN; LIN, 1998), including algorithms for re-synthesis based on AIG rewriting (MISHCHENKO; CHATTERJEE; BRAYTON, 2006). Scalability is obtained by keeping the value of K small so that logic functions can be manipulated as vectors of integers. For instance, in (MISHCHENKO; BRAYTON; CHATTERJEE, 2008) scalability is achieved by using functions of 16 or less inputs represented as binary truth-tables.

Algorithms for efficient cut computation are well known for single output cuts. Particularly, algorithms for exhaustive computation of *K*-feasible cuts were introduced by Cong (CONG; WU; DING, 1999) and Pan (PAN; LIN, 1998). Chatterjee (CHATTERJEE; MISHCHENKO; BRAYTON, 2006) introduced the concept of factor cuts, where exhaustive enumeration is avoided by making a separation between dag nodes and tree nodes in the AIG. The computation of factor cuts enables to work with cuts up to 16 inputs, which is not possible with the previous algorithms of exhaustive enumeration. All these algorithms for cut enumeration are only able to take the number K of inputs into account, not contemplating the benefits of multiple output reasoning. For example, in technology mapping using K feasible cuts, logic duplication may occur during the step of covering, which is likely a problem on a design flow.

Even though current technologies do support blocks with more than one output, such as FPGAs (HUTTON et al., 2004; COSOROABA; RIVOALLON, 2006), the entire flow is currently oriented to single output blocks, and a combination step is added in the end to try to take advantage of these multiple output elements.

1.3 Objective

The objective of this thesis is to introduce the idea of controlling the number of outputs L in *K*-feasible cuts. This way, by enumerating *KL*-cuts, we are able to deal directly with multiple output blocks throughout the logic synthesis process. Applications of *KL*-cuts may include peephole optimization (WERBER; RAUTENBACH; SZEGEDY, 2007), regularity extraction (ROSIELLO et al., 2007) and technology mapping. The use of *KL*-feasible cuts in peephole optimization is justified as an arbitrary portion of the circuit, potentially having multiple outputs, can be exchanged by another one by taking into account all signals which it affects at once. Its use in regularity extraction can be justified as many regular (logic) patterns are composed of multiple output circuits. This is especially true for arithmetic circuits, e.g. full adder and half adder library cells. Technology mapping aiming dual output blocks is already a reality (HUTTON et al., 2004; COSOROABA; RIVOALLON, 2006), and multiple output blocks should be explored, even on a standard cell flow.

1.4 Thesis Organization

The remaining of this thesis is organized as follows.

Section 2: *Technical Background* — Provides the reader with all basic and consolidated knowledge that is needed to understand the concepts presented in this work.

Section 3: *State of the Art* — Traces an evolutionary line over the technology mapping, and describes some recent works that are connected to the work presented in this document.

Section 4: *KL-Feasible Cuts* — Describes the main contribution of this work, which is the concept of *KL*-cuts, along with some algorithms for its enumeration. It also discusses some properties of specific types of *KL*-cuts.

Section 5: *Applications of KL-Cuts* — Shows a number of proposed applications for *KL*-cuts. Some of them are defined in this thesis, including mapping algorithms and a Boolean matching, and are discussed in detail. Others are just discussed more generally as possible applications, like peephole optimizations and regularity extraction.

Section 6: *Results* — Presents and discusses results of several experiments. These results are cross referenced throughout the text, so the reader can check this section while reading the other sections of this document.

Section 7: *Conclusions and Future Work* — Presents some conclusions, summarizes the contributions of this work, and discusses some possible future work.

2 TECHNICAL BACKGROUND

This section provides a review of the basics needed to understand the concepts introduced by this work. The idea of Boolean functions and Boolean networks is illustrated, and the classification of an equation into equivalence classes is explained. It explains as well how functions can be represented using different data structures. Then a manner to enumerate ways of breaking these data structures into smaller and more easily treatable ones is discussed. This is followed by a succinct revision on cell libraries and technology mapping.

2.1 Boolean Function and Boolean Network

The *Boolean set* is defined as $B = \{0, 1\}$, whose elements can be interpreted as logic values. Usually the 0 value means *false* and the 1 value means *true*. An n-dimensional Boolean set B^n is composed by all distinct Boolean vectors with length n. For example, $B^0 = \emptyset$, $B^1 = \{0, 1\}$, $B^2 = \{00, 01, 10, 11\}$, $B^3 = \{000, 001, 010, 011, 100, 101, 1100, 111\}$ *et cetera*. This way, the set B^n has 2^n elements.

A *Boolean function* $f : B^n \mapsto B$ is a function that relates every element of its domain B^n into one element of its co-domain (or image) B, i.e. each Boolean vector of length n is associated by a Boolean function either to 0 or to 1.

Boolean variables are variables in the Boolean space, i.e. they can assume values of B. A Boolean function of the form $f : B^n \mapsto B$ is a function of n variables, and each vector of B^n being an input vector defines the value of every Boolean variable. As each one of the 2^n vectors corresponds to one of the two values 0 or 1, there are 2^{2^n} different Boolean functions of n variables.

A very common representation of a Boolean function is a *truth table*. An example of truth table can be seen in Figure 2.1. Each variable is assigned to a position of the input vector, and the last column is the output value of the function for that input vector.

There are three basic Boolean operations, AND ($*$), OR ($+$) and NOT ($!$), which can be applied to Boolean values of Boolean functions. The AND and OR operations are binary operations, since they have two operands. The AND function evaluates to (or returns) 1 when all of its operands are 1, and evaluates to 0 otherwise. The OR function returns 0 when all of its operands are 0, and returns 1 otherwise. The NOT function is a unary operation, it has only one operand. When the operand is 0 the NOT function, also called inversion operation, returns 1 and vice-versa.

A *Boolean expression* is a representation of a Boolean function, and it is characterized by a particular association of Boolean variables and Boolean operations. Although an expression represents exactly one function, a function can be represented by an infinite number of different expressions. As an example, equations 2.1 and 2.2 are two possible

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.1: A truth table representation of a function.

representations of the function f depicted in figure 2.1.

$$f = (!c + (a * b)) \tag{2.1}$$

$$f = (!a*!b*!c) + (!a * b*!c) + (a*!b*!c) + (a * b) \tag{2.2}$$

Boolean operations can also be represented graphically as nodes in a graph. This way, Boolean expressions can be viewed as graphs, potentially sharing signals between them. This graphical representation of a set of Boolean functions is called a *Boolean network*. An example of a Boolean network is shown in figure 2.2.

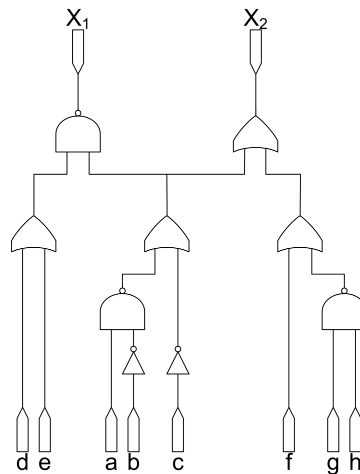

Figure 2.2: A Boolean network.

2.2 Equivalence Classes of Logic Functions

Consider the three following operations over the function f:

op1: Negation of some variables in f.

op2: Permutation of some variables in f.

op3: Negation of f.

If a function g can be derived from f by a combination of these three operations, then the function g is *NPN-equivalent* to f. The set of functions that are NPN-equivalent to the function f forms an *NPN-equivalence class* $[f]_{NPN}$, of which f is a *representative function*. Similarly, the functions that are obtained by applying operations **op1** and **op2** in f are *NP-equivalent* to f, and form an *NP-equivalence class* $[f]_{NP}$. If only operation **op2** is considered, then a *P-equivalence class* $[f]_P$ is defined.

For example, let us define the functions $f_1 = a+b$, $f_2 = !a+b$, $f_3 = !(a+!b)$, $f_4 = a+!b$. Functions f_2 and f_4 belong to the same P-equivalence class. Functions f_1, f_2 and f_4 are of the same NP-equivalence class. Finally, all of these functions belong to the same NPN-equivalence class.

Table 2.1: Number of equivalence classes under various equivalence relations (SASAO, 1999).

	0	1	2	3	4
All functions	2	4	16	256	65536
P-equivalence class	2	4	12	80	3984
NP-equivalence class	2	3	6	22	402
NPN-equivalence class	1	2	4	14	222

Table 2.1 shows the number of different equivalence classes up to $n = 4$. When n is sufficiently large, the number of equivalence classes can be approximated as follows:

$$\text{The number of P-equivalence classes is } \frac{2^{2^n}}{n!}. \tag{2.3}$$

$$\text{The number of NP-equivalence classes is } \frac{2^{2^n}}{2^n \times n!}. \tag{2.4}$$

$$\text{The number of NPN-equivalence classes is } \frac{2^{2^n}}{2^{n+1} \times n!}. \tag{2.5}$$

To categorize functions into equivalence classes is useful in the matching phase of a technology mapping. The task of the matching phase is to find sub-functions that are equivalent to one of those in the target library. As the objective is to cover the subject graph with black boxes connecting each other, the permutation operation is free. Hence the P-equivalence class is useful for any technology mapping. Moreover if the logic style used is dual rail, when all functions are generated both direct and complemented, then the negation operation is also free, so the NPN-equivalence class is the most useful.

2.3 Data Structures

A logic circuit or Boolean network may be represented by a variety of data structures. Each of them is more or less appropriate to a specific manipulation, having its particular strengths and weaknesses. An appropriate data structure is a key element to an efficient computation.

2.3.1 Directed Acyclic Graph

Graphs of operators are widely used data structures to represent the combinational part of circuits. A *Directed Acyclic Graph* (DAG) $\mathcal{G} = (V(\mathcal{G}), E(\mathcal{G}))$ is a directed graph and has no directed cycles. A node in the graph $v \in V(\mathcal{G})$ represents a logic function, such as AND or OR, a *primary input* (PI) or a *primary output* (PO). The set of directed edges $e \in E(\mathcal{G})$ arriving in the node v, $iedge(v)$, represent the inputs of v. The set of edges leaving from v, $oedge(v)$, represent the outputs of v. The *fanout* of v is defined as $|oedge(v)|$. If $\forall v \in V(\mathcal{G})$, $|iedge(v)| \leq K$, then the DAG is *K-bounded*. An example of DAG can be seen in figure 2.3, it represents the same Boolean network of figure 2.2.

Figure 2.3: A directed acyclic graph example.

A DAG is a direct translation of a Boolean network into a data structure. It represents the circuit in an integrated way, but algorithms that perform technology mapping directly over DAGs tend to suffer of scalability problems.

2.3.2 Forest of Trees

A *tree* is a particular case of DAG, in which $\forall v \in V(\mathcal{G})$, $|oedge(v)| = 1$. In other words, the fanout of every node is equal to one. In order to be able to represent any circuit as a forest of trees, every logic function that has a fanout larger than one will originate a new tree. This partitioning of the graph reduces the complexity of the subsequent technology mapping, at the cost of reducing the space of solutions, thus producing a mapping of inferior quality. A forest of trees representing the same Boolean function as before is depicted in figure 2.4.

Breaking a DAG into a forest of trees can greatly reduce the computational cost of the technology mapping process. Algorithms that achieve optimum results both in area and timing in polynomial time exist, but the optimality is only guaranteed for each tree individually. Hence, the use of trees on technology mapping can severely restrict the quality of the results.

2.3.3 And-Inverter Graph

An *And-Inverter Graph* (AIG), $\mathcal{G}$, is a specific type of DAG. Each node $v \in V(\mathcal{G})$ has either 0 or 2 incoming edges, hence it is 2-bounded. Nodes that have no incoming edges are primary inputs, and nodes having 2 inputs are AND nodes. Each edge can be complemented or not, denoting the inversion operation. Some nodes are marked as

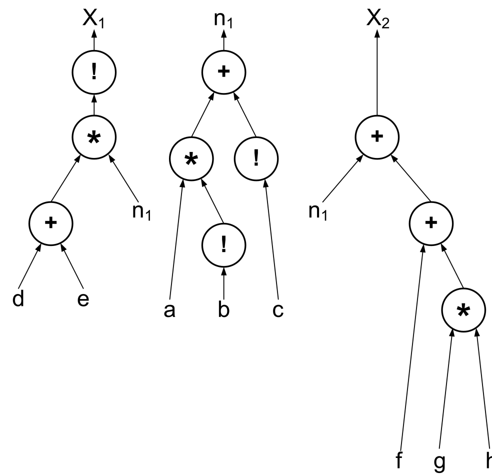

Figure 2.4: A forest of trees.

primary outputs (PO). An AIG of the same Boolean network of figure 2.2 is shown in figure 2.5.

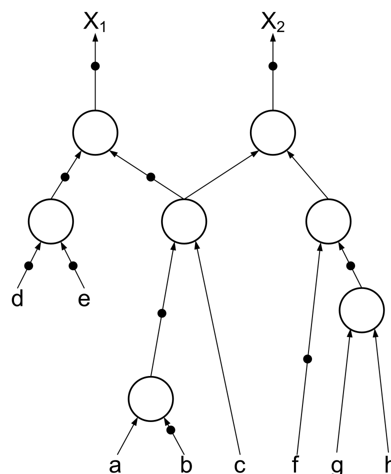

Figure 2.5: An and-inverter graph.

An AIG is a restricted DAG, though the problems of using it in technology mapping are reduced by the regularity and simplicity of its structure.

2.3.4 Binary Decision Diagram

Graphs of operators are structural representations of circuits, as they have as starting point the description of a circuit as interconnected logic operations. Another way of representing the logical behavior of a circuit is to represent directly the logic functions of each output of the circuit. A logic function is characterized by its truth table, which relates every possible input vector into an output value. Storing directly the truth table of a function is not efficient because, for a logic function having n input variables, the table has 2^n values. A more compact representation is the BDD.

Binary Decision Diagrams (BDDs) are graph representations of Boolean functions (BRYANT, 1986). A BDD is a DAG with two terminal nodes, called *0-terminal* and *1-terminal*. Each non-terminal node has an index to identify an input variable of the Boolean function and has two outgoing edges, the *0-edge* and the *1-edge*. A BDD is represented in Figure 2.6. It represents the same function f from the truth table in Figure 2.1.

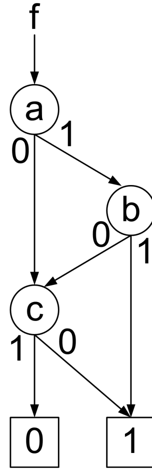

Figure 2.6: An example of a BDD.

2.4 Dag Nodes and Tree Nodes

A *dag node* is defined as a node with fanout larger than one. The nodes that are not dag ones are *tree nodes*. The set of all dag nodes in a graph $\mathcal{G}$ is represented by $\mathcal{F}$, and the set of tree nodes by $\mathcal{T}$.

A sub-graph of $\mathcal{G}$, $\mathcal{G}_{\mathcal{T}}$, induced by the nodes in $\mathcal{T}$ is a forest of trees. The root node of a tree in $\mathcal{G}_{\mathcal{T}}$ is either an input of a dag node or a PO. Consider a sub-graph $\mathcal{T}_n$ induced by a dag node n and the trees in $\mathcal{G}_{\mathcal{T}}$ that are inputs to it. $\mathcal{T}_n$ is a *factor tree*. In addition, when the root node of a tree in $\mathcal{G}_{\mathcal{T}}$ is a PO, the tree itself is also a factor tree. This way, each node n in $\mathcal{G}$ is contained in a single factor tree. In Figure 2.7, the dag nodes are shaded, and its factor trees are delimited.

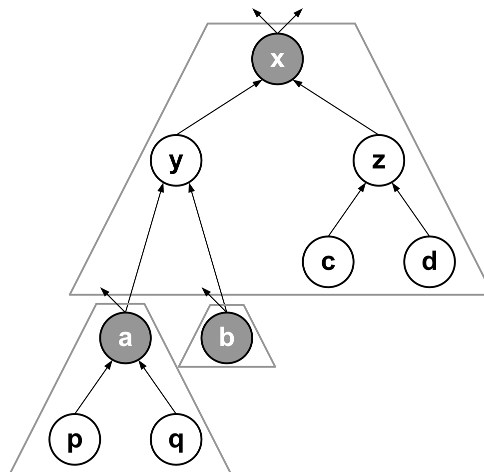

Figure 2.7: AIG illustrating dag and tree nodes (CHATTERJEE; MISHCHENKO; BRAYTON, 2006). Nodes p, q, b, c and d are primary inputs. Node x is a primary output.

A leaf of a factor tree that is not a PI has dag nodes as its inputs. A factor tree along with the dag nodes that are inputs of its leaves is called a factor leaf-DAG. In Figure 2.7, the factor leaf-DAG for the node x is its factor tree in conjunction with the nodes a and b.

2.5 K-Feasible Cuts

A *K*-feasible cut of a node n defines a subgraph, more specifically a logic cone, rooted in n, having no more than K inputs. In other words it defines a region in the graph that represents the logic function of n, using at most K variables. It is a useful tool in technology mapping, especially when targeting FPGAs, which are composed of LUTs that can implement any logic function up to a fixed number of inputs.

Formally, a *cut* of a node n is a set of nodes c such that every path between a PI and n contains a node in c. If a cut c_1 is a subset of a cut c_2, then c_1 dominates c_2. A cut is *irredundant* if it is not dominated by another cut. A *K-feasible cut* is an irredundant cut containing K or lesser nodes (PAN; LIN, 1998; CONG; WU; DING, 1999). The region defined by a *K*-feasible cut is composed by all nodes contained in a path between a node in c and the node n, including n and excluding the nodes of c.

The nature of the algorithm for enumeration of cuts is combinatorial. The combination of two cuts, where each cut is a set of nodes, is simply the union of these two sets. As each node has a set of cuts associated to it, it is of interest to define an operation that combines cuts as a Cartesian product between two sets of cuts. Notice that the simple combination of two *K*-feasible cuts does not guarantee that the resulting cut is *K*-feasible. Therefore the proposed combination operation should remove any cut that is not *K*-feasible.

Let A and B to be two sets of cuts. Let the auxiliary operation $\bowtie$ to be:

$$A \bowtie B = \{a \cup b | a \in A, b \in B, |a \cup b| \leq K\} \tag{2.6}$$

Let $\Phi_{\mathcal{K}}(n)$ to be the set of *K*-feasible cuts of $n \in \mathcal{G}$, and if n is an AND node, let n_1 and n_2 to be its inputs. Then, $\Phi_{\mathcal{K}}(n)$ is defined recursively as follows:

$$\Phi_{\mathcal{K}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PI} \\ \{\{n\}\} \cup (\Phi_{\mathcal{K}}(n_1) \bowtie \Phi_{\mathcal{K}}(n_2)) & : \text{otherwise} \end{cases} \tag{2.7}$$

The $\bowtie$ operation can also easily remove the redundant cuts, by comparing the cuts with one another, and possibly by making use of signatures (explained in section 3.3).

Table 2.2 illustrates the computation of *K*-feasible cuts. The AIG being computed is the one of figure 2.7, and the value of K is kept unlimited for the sake of the example. Notice that each node has one cut composed only by itself (called the *trivial cut*) in addition to all combinations of its fanin nodes' cuts.

Table 2.2: An example of *K*-feasible cuts computation.

Node	*K*-feasible cuts
p	$\{p\}$
q	$\{q\}$
a	$\{a\}, \{p, q\}$
b	$\{b\}$
y	$\{y\}, \{a, b\}, \{b, p, q\}$
c	$\{c\}$
d	$\{d\}$
z	$\{z\}, \{c, d\}$
x	$\{x\}, \{y, z\}, \{c, d, y\}, \{a, b, z\}$, $\{a, b, c, d\}, \{b, p, q, z\}, \{b, c, d, p, q\}$

2.6 Library

A *cell library* is a finite set of primitive logic gates, including combinational, sequential (e.g. flip-flops) and interface (e.g. drivers) elements. Here we focus on combinational cells, which are implementations of Boolean functions. This may seem too restrictive, but practical approaches of technology mapping usually deal only with the combinational portion of the circuit.

Traditionally, technology mapping algorithms rely on static pre-characterized libraries. Each cell of the library is fully characterized through exhaustive simulations, resulting in accurate information about the behavior of the cell concerning timing, power consumption and its physical area. This way, the technology mapping algorithms are restricted to use these cells in the mapping process. This approach is known as *library-based* mapping.

The quality of the final circuit is increased as the library becomes richer, i.e. a larger cell variety, both in number of functions implemented and in different flavors of each logic function (VUJKOVIC; SECHEN, 2002). However the number of different P-equivalence classes, or even NPN-equivalence classes, grows exponentially with the number of inputs. Further, the processes of electrical characterization and layout generation are extremely computing demanding, making the creation of an exhaustive library, even for a number of inputs not so large, unfeasible (SECHEN et al., 2003).

As an alternative to this duality — either to have a rich library but at a high development cost, or to have a restricted library at expense of the final quality — lies the concept of *library-free* technology mapping. The main idea is that the library is not fully designed and characterized prior to the mapping, but it is defined by means of rules, e.g. the maximum number of inputs, or the maximum number of series transistors. This reduces drastically the cost of maintaining such a large library. However, as the cells are not characterized, not even laid out, the technology mapping does not dispose of sufficient information to choose which cells to use. Thus estimation methods for each piece of information required by the mapping must be provided. These estimation methods must be fast, in order to be able to treat many cells in a short time, and accurate, in order to not mislead the mapping process. Also, once the mapper has chosen which cells must be used, they need to be properly generated, and as the variety of cells is potentially large, an automatic layout generator may be required.

2.7 Technology Mapping

Technology mapping, also known as technology binding, transforms a logic network independent from a technology into gates implemented in a technology library (HASSOUN; SASAO, 2002). It can be decomposed in three phases: *decomposition*, *pattern matching* and *covering*.

2.7.1 Decomposition

The decomposition process transforms the initial representation of the circuit into a more simple and restricted one, in order to aid the technology mapping algorithm. For example, it applies the same structural transformations in the graph representations of cells, if structural matching is used, or it breaks the graph into trees, if the technology mapping was designed to map only these data structures. This new representation, called the *subject graph* depends strongly on the future mapping strategy.

One of the tasks of the decomposition phase is to assure that each node of the subject

graph does have at least one match against the library, considering the technology mapping to be used. If this can be achieved, then at least one successful covering is guaranteed to exist.

2.7.2 Pattern Matching

Once the data structure is constructed, the pattern matcher finds a set of matches between nodes on the circuit and a predefined library. This means being able to determine whether a portion of the subject graph can be implemented by a cell in the library. Matching algorithms can be classified into two major groups: structural matching and Boolean matching (MICHELI, 1994).

Structural matching relies on the identification of common patterns. For this reason, both the subject graph and the library cells must be decomposed in the same way, so the matching process can be reduced to a graph isomorphism testing. The cells in the library may have more than one representation in the subject graph format, hence more than one representation must be maintained. Even though the problem of determining isomorphism between two graphs may be intractable, considering the size of the subgraphs used in technology mapping the computational time can be neglected.

Boolean matching relies on the identification of Boolean functions of the same equivalence class. This approach is less restrictive than structural matching, because a same function can be represented by many different graphs. It usually performs the matching using BDDs by trying different variable orderings, until a matching is found. Boolean matching is computationally more expensive than structural matching, but can lead to better results.

2.7.3 Covering

The final step, the covering, chooses a subset of the matches such that the entire network is covered and optimizing some objective function. This function is often the total area, the largest delay, the power consumption, or a composition of these.

More formally, the result of a covering is a set of cells, which have input and output signals, such that:

- Every node of the subject graph is covered by at least one cell.
- Each signal that is input of a cell is an output of another cell.
- Every cell has at least one output used by another cell as an input.

The quality of the mapping will depend significantly on the quality of the subject graph composed in the first stage of the mapping. The best algorithm executed over a poor subject graph may produce a worse result than an average mapping over a good graph. This problem is known as *structural biasing*.

As the technology mapping is a step that transforms every cell in the network, it has a major impact on the final circuit characteristics.

3 STATE OF THE ART

This section starts with a review of the evolution of technology mapping, both library-based and library-free. After that, a collection of works that are related to the research presented in this thesis is summarized.

3.1 Technology Mapping

The first methods for automatic synthesis of digital circuits had no specific algorithms for technology mapping. The synthesis process was restricted to applying a set of rules over a structure that represents the circuit, seeking some kind of optimization. The main approaches of synthesis based on rules were presented in the 1980s, by Darringer (1981) and Gregory (1988). This kind of methodology performs local optimizations, trying to reduce the cost of a region of the circuit. However, not all local optimizations lead towards global optimizations, given a particular objective function. As alternatives to the rules based system, new algorithmic solutions have emerged in order to perform the technology mapping of circuits using heuristics or even exact algorithms.

The first technology mapping algorithm, called DAGON, was proposed by Keutzer (1987). He noted a similarity between technology mapping and the tasks of a compiler. The pattern matching between sub-graphs of a circuit representation and cells of a library is a similar problem as to identify patterns between intermediate representations of a computer program and a given set of machine instructions. The subject graph used is a binary tree represented in the form of a string, and this description was consistent with the input format of the compiling engine. The structural matching and the initial representation of the circuit restrict the search space to be explored by mapping, affecting the quality of the mapped circuit. Moreover, the algorithm requires all isomorphic matches to be stored in each node of the tree, until the end of covering step. This precluded the use of very large libraries, when the number of patterns found is usually higher. It also required greater storage capacity and more time to find a solution.

In the same year, Detjens (1987) proposed the first method that actually used trees as the subject graph. In addition to this innovation, Detjens proposed the insertion of pairs of inverters in the graph. This increases the possibility of identifying new patterns in the graph (graph isomorphism), increasing the solution space. However to take advantage of this increased search space, it is necessary to create several decompositions for each element of the library. Thus, the use of a large library becomes unfeasible.

Some years later, Mailhot (1993) presented the first technology mapping algorithm that used an approach of functional verification to identify patterns. Like previous algorithms (KEUTZER, 1987; DETJENS et al., 1987), the initial DAG is partitioned into a forest of trees. However, the comparison between the sub-trees and the cells of the library

is performed by using BDDs. Since BDDs are a canonical form of representing Boolean functions, finding matches did not depend anymore on the structure of these sub-trees. However, this Boolean approach was computationally expensive, leading to limitations similar to previous approaches.

The dynamic reorganization proposed by Lehman (1995) was another alternative to minimize the dependence on the initial graph representation. In this algorithm, the decomposition phase is integrated with the pattern matching. Graphs functionally equivalent but structurally different are associated with each node of the graph, increasing the search space in order to find better solutions. Consequently, by storing many decompositions per node, the graph grows rapidly, making it impractical for large circuits.

Kukimoto (1998) proposed a method in which the mapping is executed directly over a DAG representation, and ensures optimum result in terms of speed, regardless of the initial decomposition of the graph. However, it is necessary to emphasize that the delay model, for which the optimal solution is guaranteed, ignores the load of the cell, taking into account only its propagation delay. This means that it requires a post-processing step to ensure the proper sizing of the logic cells.

Stok (1999) proposed the algorithm called *wavefront*, which is similar to Lehman's approach, but solving the scalability problem. Like Kukimoto's method, the circuit is represented by a DAG and the delay model is independent of the cell's load. To prevent the DAG to increase exponentially with the insertion of different representations for each node, the stages of decomposition, pattern matching and covering are executed concurrently in a "sliding window", called wavefront, which is tunable in terms of logic depth. This heuristic algorithm has performed well compared to its predecessors.

The state-of-the-art in library-based technology mapping is presented by Chatterjee (2006). It brings together a series of techniques used in logic synthesis, integrated and well calibrated to the benefit of the technology mapping. The essence of the mapping algorithm is the same of Kukimoto's method. The main differences are in the pattern matching, which is a Boolean matching, and in the data structure, which is an AIG. This algorithm was incorporated into an academic tool, called ABC (Berkeley Logic Synthesis and Verification Group, 2010).

In parallel with the evolution of library-based algorithms, methods based on virtual libraries have also been proposed. The first one was presented by Berkelaar (1988), and like the first methods for technology mapping, it partitions the circuit into logic cones, but does not use trees to represent them. Expressions of sums-of-products and product-of-sums are represented as graphs, using a prefixed notation. Traversing these graphs from outputs to inputs, they are partitioned into logic cells. This happens every time that a certain portion of the graph reaches a limit imposed by a set of constraints that defined the virtual library. The biggest problem with this approach is that it is a greedy algorithm. As the cuts are made top-down, the logic depth of what is below is unknown. So, it is not possible to guarantee a solution with minimum logic depth or with minimum number of cells.

Abouzeid (1993) proposed a new approach to generating cells, motivated by the possibility of using a large number of logic cells. In this method, the initial DAG is also partitioned into trees, but these are n-ary trees, as each node can have n child nodes. The representation as an n-ary tree decreases the dependence on the initial graph, allowing change of structure in a given set of nodes. Based on this representation, cuts are made each time a node that exceeds any of the restrictions is found. Although the cuts are generated from inputs to outputs, they are made in a greedy way, not contemplating logic

depth minimization.

Liem (1992) proposed a method based on a strategy called constructive matching. Unlike previous approaches that considered only maximum values for chains of transistors, this method considered the number of inputs and logic depth of a cell. These two additional restrictions are imposed to ease post-mapping stages, because the used cells are smaller. This also restricts the number of possible matchings, reducing the complexity of the problem. One problem is the algorithm dependence on the initial structure of the circuit, considering that the circuit is represented by binary trees. Another problem is the storage of the matchings up until the covering, which also precluded the method for mapping large circuits.

The method presented by Reis (1999) proposes a different approach in library-free. The representation of each logic cone is made by a special type of BDD, called Terminal-Suppressed Binary Decision Diagram (TSBDD). An interesting property of this structure is the direct association of the arcs of the BDD to transistors. However, this representation faces the same problems of representation by trees. The major contribution of this method was the use of dynamic reordering on the initial representation of the circuit.

Later on, Jiang (2001) proposed the Odd-level Transistor Replacement (OTR) method. This method works directly on a graph representing the electrical diagram at transistor level of a circuit, so it depends on an already mapped structure. The goal of the algorithm is to select which gates can be collapsed in order to achieve a better performance. Like most of the methods, it also depends strongly on the initial decomposition of the circuit.

A latter strategy for mapping based on virtual libraries, and using trees as a subject description, was presented by Correia (2004), originating the ELIS tool. This method uses n-ary trees to represent the cones of a logic circuit. The main advantage of this algorithm is that it considers several decompositions of sub-trees dynamically (at a low computational cost), leading to a minimum coverage using dynamic programming. The major limitation comes from the use of by trees, which prevent a broader view of the circuit.

Marques (2007) proposes the VIRMA algorithm. VIRMA performs the mapping over a DAG, aiming the reduction the circuit delay. The library uses a maximum number of series transistor, but considering the lower bound (SCHNEIDER et al., 2005) in a topologically non-complementary implementation of CMOS cells. It also uses a sliding window to reduce the complexity, and achieves a reasonable scalability comparing to its predecessors.

All methods discussed above have limitations imposed by how they address the problem of technology mapping. In general, the application of heuristics is necessary to ensure the tractability of the problem. As an example of heuristic, the mapping for minimum area in a DAG is NP-complete, but if the DAG is partitioned into trees and each tree mapped independently, optimum coverings for each tree can be found in linear time. Regarding a mapping for minimal delay, it can be achieved on a DAG mapping depending on the delay model. However, it should be noted that these models are not precise enough to ensure proper sizing of the circuit. Thus, a further step of sizing is necessary. There are methods that try to solve the problem of mapping and sizing simultaneously. The algorithm proposed by Karandikar (2004) is an example of it. It finds good results in polynomial time using more sophisticated delay models associated to some heuristics.

3.2 DAG-Aware AIG rewriting

AIG rewriting is a greedy algorithm for minimizing the number of nodes of an AIG. It iteratively selects subgraphs and replaces them with pre-computed logically equivalent subgraphs.

The algorithm needs a hash table of pre-computed graphs, for all functions with up to a certain number of inputs inputs. The authors used functions with up to four inputs. So, first of all, a series of AIG subgraphs are computed for every one of the 222 different NPN-equivalence classes with up to four inputs.

Once the hash table is set up, the algorithm traverses the AIG in topological order, from inputs to outputs. For each node, all 4-feasible cuts are enumerated. Each 4-feasible cut is matched with the pre-computed graphs from the hash-table. Cuts that reduce the number of nodes without increasing the height of the region, or cuts that add shared nodes are sought. After trying all available subgraphs for a node, the one that leads to the greatest improvement replaces the original cut.

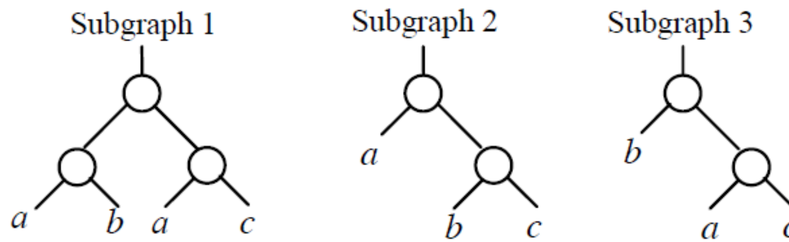

Figure 3.1: Different AIG structures for function $f = a * b * c$ (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

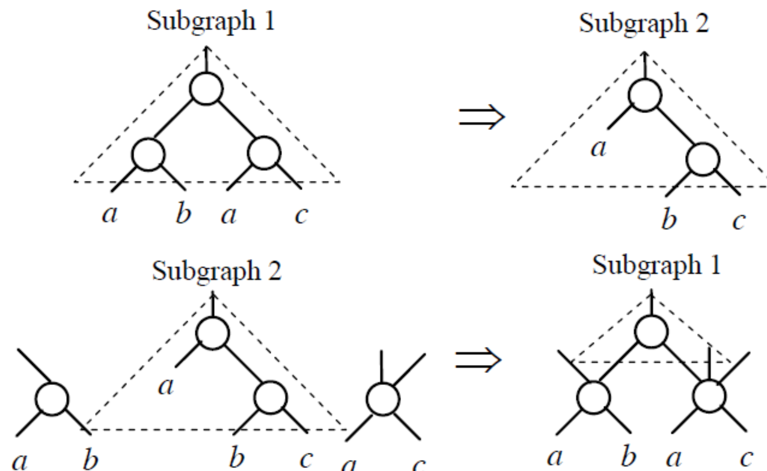

Figure 3.2: Two cases of AIG rewriting of a node (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

An example follows. Figure 3.1 shows three AIG representations of the function $f = a*b*c$, which were pre-computed and stored in a hash-table. Figure 3.2 contains two examples of AIG rewriting. In the upper example subgraph 1 was detected and replaced by subgraph 2, causing a reduction of one node. The lower example shows, besides the marked cut, two more nodes that are already elsewhere on the graph. In this case, subgraph 2 is detected and replaced by subgraph 1, a seemingly useless subgraph since it

is redundant. However the sharing of nodes caused this replacement to reduce one node in the graph.

After the entire AIG is traversed, a second type of AIG rewriting takes place, called refactoring. This algorithm has a heuristic that chooses one large cut for each AIG node. Refactoring a cut is performed by extracting the Boolean function of the cut and running an equation factorization algorithm, which is converted back to an AIG representation, possibly replacing the original cut. The change is accepted if there is a reduction of the number of nodes.

A third step consists in balancing the AIG structure. The authors suggest a script that traverses the structure 10 times, as follows: *b, rw, rf, b, rw, rwz, b, rfz, rwz, b*. In the abbreviated form, *b* stands for balancing, *rw/rf* stand for AIG rewriting and refactoring, and *rfz/rwz* is the same, but with zero improvement replacements permitted.

The authors claim that this approach leads to a reduction of area in the order of 10% and 5% gains in delay, while the runtime is reduced by a factor ranging between 7 and $\sim$1000, when comparing with certain scripts of MVSIS (MVSIS Group, 2010) and SIS (SENTOVICH et al., 1992).

3.3 Using Signatures on Cut Computation

The use of signatures on cut computation has been proposed by Mishchenko (2007). Its use speeds up the process, and does not affect the final result.

A signature, $sign(c)$, of a cut c is an M-bit integer. It is suggested by the authors to use M as the number of bits that compose a word of the target processor. Every node $n \in c$ has an ID. The signature is computed by bitwise OR operations, for each node contained on the cut, as seen on equation 3.1.

$$sign(c) = \underset{n \in c}{\mathrm{OR}}\, 2^{(\mathrm{ID}(n)\ mod\ M)} \tag{3.1}$$

Testing cut properties is much faster with signatures than testing the actual cuts. Although the use of signatures cannot avoid completely the computation over the real cuts, they are able to reduce it drastically.

If cuts c_1 and c_2 are equal then $sign(c_1) = sign(c_2)$. Hence if the signatures are different, so are the cuts. If the signatures are equal, then the cuts must be tested for equality.

If a cut c_1 dominates a cut c_2 then all the 1s in $sign(c_1)$ are contained in $sign(c_2)$. This way, if $sign(c_1)\,\mathrm{AND}\,sign(c_2) \neq sign(c_1)$ then c_1 does not dominate c_2. Otherwise, the cuts must be tested for dominance.

If $c_1 \bowtie c_2$ is a *K*-feasible cut then $|sign(c_1)\ \mathrm{OR}\ sign(c_2)| \leq K$. Here $|s|$ denotes the number of 1s on the binary representation of s. So, if $|sign(c_1)\ \mathrm{OR}\ sign(c_2)| > K$ then $c_1 \bowtie c_2$ is not *K*-feasible, otherwise its *K*-feasibility must be tested.

An example follows. Let $M = 8$. Cut c_1, having the nodes with ids 32, 68 and 69 would have $sign(c_1) = 00110001$. A second cut c_2 with nodes having ids 32, 68 and 70 would have $sign(c_2) = 01010001$. It can be inferred that neither c_1 dominates c_2 nor c_2 dominates c_1, without having to actually compare the cuts. If c_3 is a cut composed by nodes having ids 36, 64 and 69, then $sign(c_3) = sign(c_1) = 00110001$. However $c_1 \neq c_3$, which shows that the comparison of the cuts is sometimes necessary.

3.4 Factor Cuts

Factor cuts (CHATTERJEE; MISHCHENKO; BRAYTON, 2006) are a collection of *K*-feasible cuts, grouped in two categories, *local cuts* and *global cuts*. The definition of these groups can vary according to the factorization scheme. However, the idea is to construct these two sets of cuts in order to be able to expand them generating a (possibly complete) set of *K*-feasible cuts. Observe that in this context the term factorization has a distinct meaning from factorization of Boolean equations.

In short, factor cuts allow algorithms that were conceived to use *K*-feasible cuts to work without the need of enumerating all cuts of every node. Only factor cuts need to be computed, and further calculation can be executed on-the-fly as more cuts are needed.

The expansion process can be explained as follows. Let c to be a global cut of a node n, and let c_i to be a local cut of a node i belonging to c. If e is a cut defined as $e = \bigcup_i c_i$, and e is *K*-feasible, then e is a *1-step expansion* of n. The set of cuts obtained expanding the cut c is defined as 1-step(c).

$$\text{1-step}(c) = \{e | e \text{ is a 1-step expansion of } c\} \tag{3.2}$$

In Figure 2.7, expanding the node a in the global cut $\{a, b, z\}$ by its local cut $\{p, q\}$, we get the cut $\{p, q, b, z\}$, which is therefore a 1-step expansion of $\{a, b, z\}$.

3.4.1 Complete Cut Factorization

When using *complete factorization*, the local cuts are the *tree cuts*, and the global cuts are the *reduced cuts*. The complete factorization has an interesting property: any *K*-feasible cut can be generated by 1-step expansion.

3.4.1.1 Tree Cuts

The tree cuts of n are cuts only involving nodes within its factor tree.

Let $\Phi_{\mathcal{KT}}(n)$ be the set of tree cuts of a node n. Define the auxiliary function $\Phi^{\dagger}_{\mathcal{KT}}(n)$ as follows:

$$\Phi^{\dagger}_{\mathcal{KT}}(n) = \begin{cases} \emptyset & : n \in \mathcal{F} \\ \Phi_{\mathcal{KT}}(n) & : \text{otherwise} \end{cases} \tag{3.3}$$

Then, $\Phi_{\mathcal{KT}}(n)$ is defined by:

$$\Phi_{\mathcal{KT}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PI} \\ \{\{n\}\} \cup (\Phi^{\dagger}_{\mathcal{KT}}(n_1) \bowtie \Phi^{\dagger}_{\mathcal{KT}}(n_2)) & : \text{otherwise} \end{cases} \tag{3.4}$$

So, $\Phi_{\mathcal{KT}}(n)$ is the subset of $\Phi_{\mathcal{K}}(n)$ that is composed only of nodes from the factor tree of n.

For example, in Figure 2.7, $\Phi_{\mathcal{KT}}(x) = \{\{x\}, \{y, z\}, \{y, c, d\}\}$.

3.4.1.2 Reduced Cuts

The set of reduced cuts of a node n, $\Phi_{\mathcal{KR}}(n)$, is defined as:

$$\Phi_{\mathcal{KR}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PI} \\ \{\{n\}\} \cup ((\Phi_{\mathcal{KR}}(n_1) \bowtie \Phi_{\mathcal{KR}}(n_2)) \backslash \Phi_{\mathcal{KT}}(n)) & : \text{otherwise} \end{cases} \tag{3.5}$$

The formula of $\Phi_{\mathcal{KR}}(n)$ is very similar to $\Phi_{\mathcal{K}}(n)$, except that the tree cuts of n are recursively removed. Because of that $\Phi_{\mathcal{KR}}(n)$ is much smaller than $\Phi_{\mathcal{K}}(n)$.

In Figure 2.7, $\Phi_{\mathcal{KR}}(x) = \{\{x\}, \{a, b, z\}\}$.

3.4.2 Partial Cut Factorization

The *partial factorization* scheme does not allow the generation of a complete set of *K*-feasible cuts by 1-step expansion as the complete factorization scheme does, although it is much faster and in practice produces good results. For partial factorization, the local cuts are the *leaf-dag cuts*, and the global cuts are the *dag cuts*.

3.4.2.1 Leaf-dag Cuts

The leaf-dag cuts of a node n are cuts only involving nodes of its factor leaf-DAG.

Let $\Phi_{\mathcal{KL}}(n)$ be the set of leaf-dag cuts of a node n. Define the auxiliary function $\Phi^{\dagger}_{\mathcal{KL}}(n)$ as follows:

$$\Phi^{\dagger}_{\mathcal{KL}}(n) = \begin{cases} \{\{n\}\} & : n \in \mathcal{F} \\ \Phi_{\mathcal{KL}}(n) & : \text{otherwise} \end{cases} \tag{3.6}$$

Then, $\Phi_{\mathcal{KL}}(n)$ is defined by:

$$\Phi_{\mathcal{KL}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PI} \\ \{\{n\}\} \cup (\Phi^{\dagger}_{\mathcal{KL}}(n_1) \bowtie \Phi^{\dagger}_{\mathcal{KL}}(n_2)) & : \text{otherwise} \end{cases} \tag{3.7}$$

Leaf-dag cuts are conceptually similar to the tree cuts. The difference lies at the fact that leaf-dag cuts include also the dag nodes that are inputs to the factor tree of n.

For example, in Figure 2.7, $\Phi_{\mathcal{KL}}(x) = \{\{x\}, \{y, z\}, \{a, b, z\}, \{y, c, d\}, \{a, b, c, d\}\}$. Notice that $\{a, b, z\}$ and $\{a, b, c, d\}$ are not tree cuts of x.

3.4.2.2 Dag Cuts

Let $\Phi_{\mathcal{KD}}(n)$ define the set of dag cuts of the node n:

$$\Phi_{\mathcal{KD}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PI} \\ (\Phi_{\mathcal{KD}}(n_1) \bowtie \Phi_{\mathcal{KD}}(n_2)) & : n \in \mathcal{T} \\ \{\{n\}\} \cup (\Phi_{\mathcal{KD}}(n_1) \bowtie \Phi_{\mathcal{KD}}(n_2)) & : \text{otherwise} \end{cases} \tag{3.8}$$

This way, $\Phi_{\mathcal{KD}}(n)$ will only contain dag nodes and primary inputs. The number of dag cuts is much smaller than reduced cuts, but still allows us to capture much of the reconvergence in the network.

In Figure 2.7, $\Phi_{\mathcal{KD}}(x) = \{\{x\}, \{a, b, c, d\}, \{p, q, b, c, d\}\}$.

3.5 TEMPLATE Boolean Matching Method

The TEMPLATE system (TEchnology Mapping PLATform) (HINSBERGER; KOLLA, 1998) is a method for Boolean matching of functions, which tries to find whether functions belong to the same equivalence class.

The method is based on the definition of a canonical representative function $R[f]$ for each equivalence class $[f]$. Thus, the matching of a function against a library can be performed as follows: first the canonical representative function is computed for each function in the library; then the representative for the target function is computed and a direct comparison takes place, if $R[f_1] = R[f_2]$ then f_1 is a match of f_2.

A function is defined by its truth table, which can be represented as a bit string. When looking at a bit string as an unsigned integer representation, there is an inherent ordering.

This way, the two functions can be compared and classified as larger, smaller or equal by comparing the integers formed by their bit string representations.

So the representative function $R[f]$ of $[f]$ can be defined as the largest function in $[f]$, as said in equation 3.9.

$$R[f] = \max_{g \in [f]} g \tag{3.9}$$

The most intuitive way of finding $R[f]$ would be trying all operations allowed in the target equivalence class. For example, figure 3.3 shows functions of three variables obtained by all possible permutations of the inputs, hence considering a P-equivalence class, in the format of a tree. In the figure, the variables are represented by the numbers 1, 2 and 3. Each node contains a variable ordering assigned. In the $S_{0,3}$ line there is no ordering assigned to any variable, in the line $S_{1,3}$ the first variable is assigned, and so on.

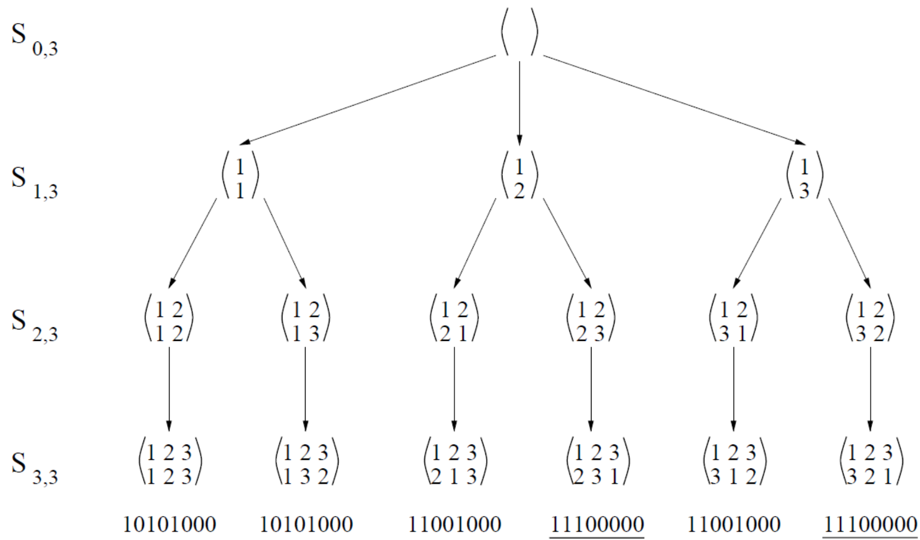

Figure 3.3: Naive approach for computing $R[f]_P$ (HINSBERGER; KOLLA, 1998).

One interesting property can be visualized with the help of figure 3.4. If the variable ordering is defined up to the k-th variable, no matter the ordering of the subsequent variables, the first 2^k lines of the truth table are already defined. This is because every variable having an index larger than k has an equal value (zero) on these first 2^k lines.

$x_n \dots x_{k+1}$	$x_k \dots x_1$	$f(x_1 \dots x_n)$
0	$0 \dots 0$	T_0
	$\vdots$	$\vdots$
	$1 \dots 1$	T_{2^k-1}
$0 \dots 1$	$0 \dots 0$	T_{2^k}
$\vdots$	$\vdots$	$\vdots$
$1 \dots 1$	$1 \dots 1$	T_{2^n-1}

Figure 3.4: A generic view of a truth table.

With this property in mind, at each node of the tree some of the first values of the function can be evaluated. As the most significant bits play a more important role on

determining relations of equality and ordering, some branches of the tree are already known not to produce the largest integer, hence some branches are not computed any further. Following the same example, figure 3.5 shows a reduction on the computation of $R[f]_P$ when comparing to 3.3. This way the computation of the whole tree is avoided, by not reaching the not maximal leaves.

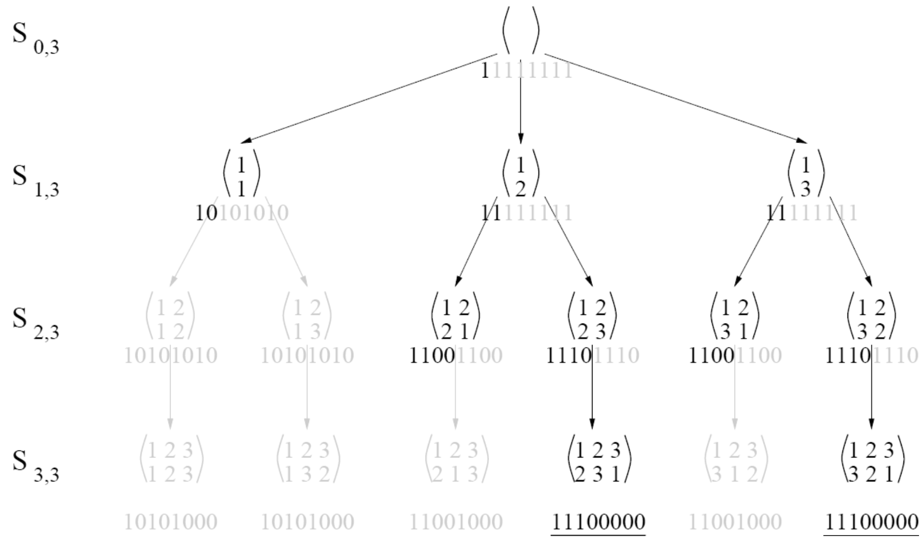

Figure 3.5: Reducing search space by cutting non-maximal branches (HINSBERGER; KOLLA, 1998).

Although reduction is achieved by cutting branches, every branch producing a maximal integer is still reached, even though only one could provide the correct representative. To reduce even further the computational effort, information of variable symmetry can be used. Two variables a and b of a function f are *symmetrical* when they can be exchanged without changing the result of the function. More formally, if $f(a, b) = f(b, a)$ then a and b are symmetrical. The set of variables that are symmetrical between them defines a symmetry class.

From the definition of variable symmetry, it is deductible that not every variable should be tested in every position, but only one variable of each symmetry class. This is because if the only difference between two branches is that the position of two symmetric variables is exchanged, then these two branches produce the same integers in their leaves. The same example now is shown in figure 3.6, but taking advantage of the fact that variables 2 and 3 are symmetrical.

The authors also define a generalization for the NPN-equivalence class case. Instead of considering only permutations, the tree is constructed by also considering inversions in the variables. This covers the NP-equivalence class. But as the tree is constructed twice, one for the direct function and another for the inverted function, the NPN-equivalence class case is covered.

The authors claim that this algorithm is able to manage about 10^6 functions per second, using an HP 735/125. The tests were executed over the 1989 MCNC benchmark circuits, and they did not provide the average size of the cuts, which is the average number of inputs of the functions treated.

Other authors (DEBNATH; SASAO, 2004) have improved this method to make it faster. The difference on this latter approach is that the entire set of possible negations and permutations is pre-computed and stored in a hash-table. According to the authors,

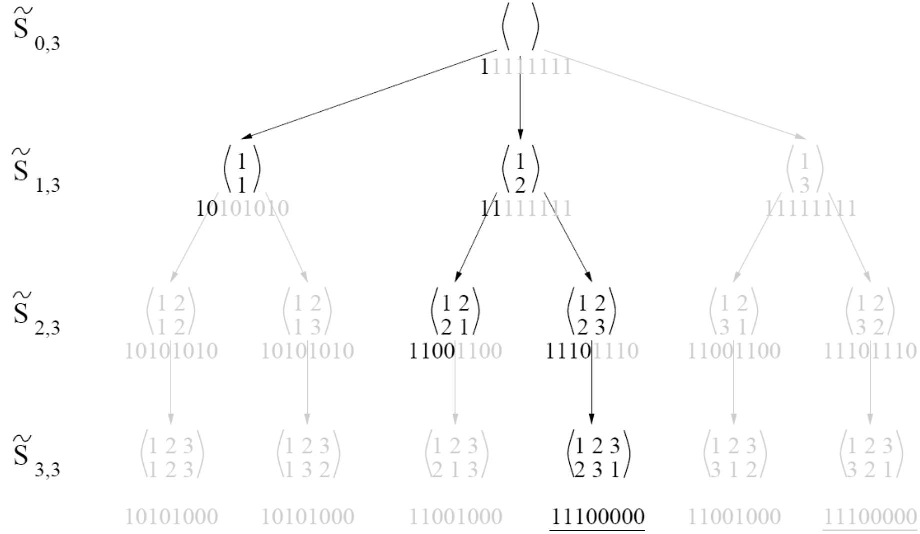

Figure 3.6: Reducing search space by using symmetry (HINSBERGER; KOLLA, 1998).

once this hash-table is computed, the matching phase presents a speed-up of two orders of magnitude, at expense of using much more memory. Hash-tables for functions with up to seven variables consume 140 megabytes of memory.

3.6 Area Flow Covering

The area flow covering algorithm (MANOHARARAJAH; BROWN; VRANESIC, 2006) uses a dynamic programming approach, and finds a solution in time proportional to the number of nodes times the average number of *K*-feasible cuts per node. An important characteristic of the algorithm is that it can be performed iteratively, and each execution of the algorithm uses information from the last one.

The area flow of a node, $\text{AF}(n)$, is an estimate of the area used up to the generation of that node, and is defined in equation 3.10. In this equation c_n is the cut rooted in n that produces the smallest $\text{AF}(n)$. It is called the *best cut* of n.

$$\text{AF}(n) = \frac{\text{Area}(c_n) + \sum_i \text{AF}(\text{Input}_i(c_n))}{\text{Fanout}_{est}(n)} \tag{3.10}$$

The sum of the area flows of the output nodes of a circuit is therefore an estimate for the area of the entire circuit.

The reason why the area flow is an estimate of the area, and not the actual area, is because the fanout of each node can only be determined after the covering of the circuit. Therefore, each iteration can take in consideration the fanout of the nodes in the previous iteration.

The fanout estimation of a node, $\text{Fanout}_{est}(n)$, used in the computation of the area flow is a weighted average between the last estimation, $\text{Fanout}'_{est}(n)$, and the fanout of the last iteration, $\text{Fanout}(n)$. According to the authors, values of α between 1.5 and 2.5 produce the best area results.

$$\text{Fanout}_{est}(n) = \frac{\text{Fanout}'_{est}(n) + \alpha\,\text{Fanout}(n)}{1 + \alpha} \tag{3.11}$$

After an iteration, the graph is fully covered. Every cell, or block, has inputs and outputs. This way, the fanout of a node $\text{Fanout}(n)$ is defined as the number of occurrences

of that signal as an input of another block. The use of a signal as a PO is also counted. At the first iteration, the covering to be considered is one cell per node of the graph, so the fanout estimation of a node is its actual fanout on the graph.

At each iteration, the graph is traversed twice. The first traversal is performed from inputs to outputs, and computes the area flow of each node. Then a traversal from outputs to inputs takes place, recursively choosing the cuts with minimal area flow. The output nodes of the chosen cuts are said to be the visible nodes, and the number of times that node is used is the definition of Fanout(c). For the nodes that are not visible, i.e. fanout equal to zero, the authors recommend using a fanout value of one.

Table 3.1 shows the effect of successive iterations over the benchmark circuit *s38584*, with a mapping oriented to an FPGA having LUTs with up to four inputs and using $\alpha = 2$. The process of mapping to an FPGA allows any cut that do not exceed the number of inputs of the LUTs to be used on the final mapping, and considers that the area of each cut is unitary. The table reports the number of LUTs used (# cuts) and the area flow seen from the primary outputs (AF) for each iteration. It is noticeable that the area flow approximates to the area as iteration continues. This is due to the correction of the fanout estimation on each iteration. As the area flow approaches the true value of area, the algorithm is able to reduce area even further, because it actually tries to reduce the total area flow of the circuit. The authors say that for most circuits the improvements cease at eight iterations.

Table 3.1: Effect of iterations in area flow.

Iteration	# cuts	AF
1	3874	3337.77
2	3831	3568.19
3	3820	3708.66
4	3818	3774.60
5	3818	3801.44
6	3818	3811.99
7	3818	3815.93
8	3818	3817.31

The first iteration also makes sure that the covering has a minimum depth, i.e. the maximum number of cuts in a path. All subsequent iterations minimize area without increasing the depth of the circuit, eventually augmenting the depth of a path that had a slack. Anyhow the implementation considered in this thesis, and subsequently the results shown in section 6.3, aims only area minimization and is unaware of the circuit depth.

4 KL-FEASIBLE CUTS

In this section the notion of *KL-feasible cuts*, or simply *KL*-cuts, is introduced. Initially, the notion of backcuts and *L*-feasible backcuts is shown. Some conceptual variations on the computation of *KL*-cuts are also discussed, and algorithms are sketched to ease the understanding of the generation process.

Cuts are an efficient way of representing a region of an AIG regarding one signal generation. However, when it comes to multiple output regions multiple cuts would be needed. To overcome this limitation, the proposed *KL*-cuts are subgraphs which not only have a limited and well controlled number of inputs, but these same properties are extended to the outputs.

A *KL-cut* defines a sub-graph $\mathcal{G}_{\mathcal{KL}}$ of $\mathcal{G}$ which has no more than K inputs and no more than L outputs. It is represented as two sets of nodes $\{\mathcal{G}_{\mathcal{K}}, \mathcal{G}_{\mathcal{L}}\}$: being $\mathcal{G}_{\mathcal{K}}$ the set of inputs and $\mathcal{G}_{\mathcal{L}}$ the set of outputs. If a node n belongs to a path between $n_K \in \mathcal{G}_{\mathcal{K}}$ and $n_L \in \mathcal{G}_{\mathcal{L}}$, and $n \notin \mathcal{G}_{\mathcal{K}}$, then n is contained in $\mathcal{G}_{\mathcal{KL}}$. Notice that all nodes in $\mathcal{G}_{\mathcal{L}}$ are contained in $\mathcal{G}_{\mathcal{KL}}$. However, $\mathcal{G}_{\mathcal{KL}}$ does not contain any node of $\mathcal{G}_{\mathcal{K}}$ (MARTINELLO et al., 2009, 2010).

A *KL*-cut is said to be *complete* when all the following conditions are met:

c1: Every path between a PI and a node $n_L \in \mathcal{G}_{\mathcal{L}}$ contains a node in $\mathcal{G}_{\mathcal{K}}$;

c2: Every path between a node contained in $\mathcal{G}_{\mathcal{KL}}$ and a PO contains a node in $\mathcal{G}_{\mathcal{L}}$;

c3: No *KL*-cut defined by a subset of $\mathcal{G}_{\mathcal{K}}$ and the same $\mathcal{G}_{\mathcal{L}}$ is complete;

c4: No *KL*-cut defined by the same $\mathcal{G}_{\mathcal{K}}$ and a subset of $\mathcal{G}_{\mathcal{L}}$ is complete.

In essence, a *KL*-cut defines a region of a graph, which have at most K inputs and at most L outputs. It is represented by the set of inputs and the set of outputs. All nodes between the set of inputs and the set of outputs, including the set of outputs but excluding the set of inputs, are “inside” the delimited region.

4.1 L-Feasible Backcuts

In order to be able to construct *KL*-cuts, another structure must be defined, so the idea of backcuts is introduced.

The algorithms for computing cuts work from inputs to outputs. Computing *KL*-cuts involves computing backward cuts — or *backcuts* — from outputs to inputs. The proposed backcuts are quite similar to cuts. However, instead of representing a set of nodes that can generate n, they represent a set of nodes that are influenced by n.

A backcut of a node n is a set of nodes c such that every path between n and a PO contains a node in c. If a backcut c_1 is a subset of a backcut c_2, then c_1 dominates c_2. A backcut is irredundant if it is not dominated by another backcut. An *L*-feasible backcut is an irredundant backcut containing L or lesser nodes (MARTINELLO et al., 2009, 2010).

Let us keep the definition of the operation $\bowtie$, only changing K for L, as shown in equation 4.1. For convenience, let us define another operator, as seen in equation 4.2. This attribution can be made since the $\bowtie$ operation is commutative.

$$A \bowtie B = \{a \cup b | a \in A, b \in B, |a \cup b| \leq L\} \tag{4.1}$$

$$\mathop{\bowtie}_{i=m}^{n} x_i = x_m \bowtie \ldots \bowtie x_n \tag{4.2}$$

Let $\Phi_{\mathcal{L}}(n)$ be the set of *L*-feasible backcuts of n, and let n_i to be the i-th node connected to its output. We define $\Phi_{\mathcal{L}}(n)$ as:

$$\Phi_{\mathcal{L}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PO} \\ \{\{n\}\} \cup (\bowtie_i \Phi_{\mathcal{L}}(n_i)) & : \text{otherwise} \end{cases} \tag{4.3}$$

As an example, in Figure 4.1, $\Phi_{\mathcal{L}}(p) = \{\{p\}, \{r, s\}, \{s, t\}\}$.

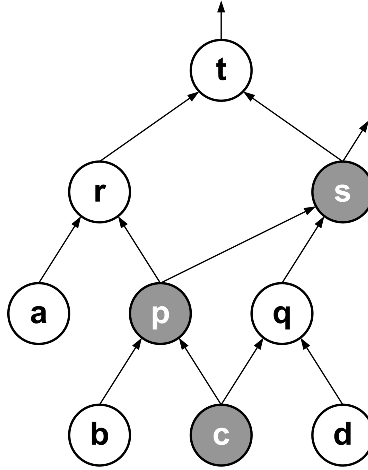

Figure 4.1: AIG demonstrating backcut factorization. Nodes a, b, c and d are primary inputs. Nodes s and t are primary outputs.

Table 4.1 shows an example of *L*-feasible cuts enumeration. The AIG used is the one represented in figure 4.1, and the value of L was kept unlimited. Each node has the backcut containing itself only (called *trivial backcut*) along with the combination of backcuts of its fanout nodes. When the node's fanout is one, it inherits the fanout's cuts. Notice that the backcuts $\{q, r, s\}$ and $\{q, s, t\}$ are redundant, since they are dominated by the cuts $\{r, s\}$ and $\{s, t\}$ respectively, and hence are not present in $\Phi_{\mathcal{L}}(c)$.

4.1.1 Factor Backcuts

As it can be done when dealing with cuts, backcuts can be factored into two groups: global and local backcuts. We propose a factorization, similar to the partial cuts factorization scheme, for backcuts, and similarly to the precursor scheme the proposed algorithm cannot generate every *L*-feasible backcut by 1-step expansion. The definition is as follows.

Table 4.1: An example of *L*-feasible backcuts computation.

Node	*L*-feasible backcuts
t	$\{t\}$
r	$\{r\}, \{t\}$
a	$\{a\}, \{r\}, \{t\}$
s	$\{s\}$
p	$\{p\}, \{r, s\}, \{s, t\}$
b	$\{b\}, \{p\}, \{r, s\}, \{s, t\}$
q	$\{q\}, \{s\}$
c	$\{c\}, \{p, q\}$, ~~$\{q, r, s\}$~~ , ~~$\{q, s, t\}$~~ , $\{p, s\}, \{r, s\}, \{s, t\}$
d	$\{d\}, \{q\}, \{s\}$

Let $\Phi^{\dagger}_{\mathcal{LL}}(n)$ to be an auxiliary function:

$$\Phi^{\dagger}_{\mathcal{LL}}(n) = \begin{cases} \{\{n\}\} & : n \in \mathcal{F} \\ \Phi_{\mathcal{LL}}(n) & : \text{otherwise} \end{cases} \tag{4.4}$$

Let $\Phi_{\mathcal{LL}}(n)$ define the set of local backcuts of the node n:

$$\Phi_{\mathcal{LL}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PO} \\ \{\{n\}\} \cup (\underset{i}{\bowtie} \Phi^{\dagger}_{\mathcal{LL}}(n_i)) & : \text{otherwise} \end{cases} \tag{4.5}$$

For example, in Figure 4.1, $\Phi_{\mathcal{LL}}(c) = \{\{c\}, \{p, q\}, \{p, s\}\}$.
And let $\Phi_{\mathcal{LD}}(n)$ denote the set of global backcuts of the node n:

$$\Phi_{\mathcal{LD}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PO} \\ \{\{n\}\} \cup (\underset{i}{\bowtie} \Phi_{\mathcal{LD}}(n_i)) & : n \in \mathcal{F} \\ \underset{i}{\bowtie} \Phi_{\mathcal{LD}}(n_i) & : \text{otherwise} \end{cases} \tag{4.6}$$

In Figure 4.1, $\Phi_{\mathcal{LD}}(c) = \{\{c\}, \{p, s\}, \{s, t\}\}$.

This definition allows the local backcuts to contain only nodes belonging to its factor leaf-DAG, and let the global backcuts to transgress the factor leaf-DAG barriers, allowing the reconstruction of many of the *L*-feasible backcuts by 1-step expansion. The expansion of backcuts works in the same way as for cuts. The global backcuts have their nodes expanded by the local backcuts. Some quantitative results are shown in section 6.1.

As an example of 1-step expansion, consider the backcut $\{p, s\}$ in Figure 4.1. Expanding the node p by its local backcut $\{r, s\}$, we get to the backcut $\{r, s\}$. Thus, $\{r, s\}$ is a 1-step expansion of $\{p, s\}$.

4.2 KL-Cuts Generation Algorithm

The objective of this algorithm is to find *KL*-cuts that have shared nodes on the generation of more than one output, that is, nodes that belong to *K*-feasible cuts of more than one output.

Figure 4.2 shows a pseudo-code for *KL*-cuts enumeration. It starts enumerating all *K*-feasible cuts and all *L*-feasible backcuts of the circuit. Each computed backcut generates a set of *KL*-cuts. The function COMBINEKCUTS combines the *K*-feasible cuts of the nodes

belonging to the current backcut d. Let d_i to be a node of d. Let $p = \bowtie_i \Phi_{\mathcal{K}}(d_i)$. This way, p is a set of input groups p_i, and each one defines a *KL*-cut $\{p_i, d\}$. Nevertheless, not every resulting *KL*-cut is complete, because condition **c2** is not assured. So, the function CHECKANDFIX adds nodes to the set of outputs in order to make the *KL*-cut complete, or else discards the *KL*-cut. If a node n_{KL} belonging to $\mathcal{G}_{\mathcal{KL}}$ has as output a node that does not belong to $\mathcal{G}_{\mathcal{KL}}$, then $n_{\mathcal{KL}}$ must be added to $\mathcal{G}_{\mathcal{L}}$. If $\mathcal{G}_{\mathcal{L}}$ still have no more than L nodes, $\mathcal{G}_{\mathcal{KL}}$ is a complete *KL*-cut, otherwise it is discarded. In this implementation the CHECKANDFIX function also discards *KL*-cuts that are not connected, as its partitions would most likely appear again as different cuts.

Particularly when $L = 2$, the connectivity test can be avoided. When combining the cuts from the two output nodes, only a pair of cuts that have a common node should be combined. This way, only connected graphs can be formed, and the testing for connectivity is skipped, speeding up the process while producing exactly the same result.

```
1: function COMPUTEKLCUTS(K, L, aig)
2:     kcuts ← COMPUTEKCUTS(aig, K)
3:     lcuts ← COMPUTELCUTS(aig, L)
4:     klcuts ← ∅
5:     for all lcut in lcuts do
6:         p ← COMBINEKCUTS(lcut)
7:         for all pi in p do
8:             klcut ← CREATEKLCUT(pi, lcut)
9:             if CHECKANDFIX(klcut) then
10:                klcuts.add(klcut)
11:            end if
12:        end for
13:    end for
14:    return klcuts
15: end function
```

Figure 4.2: Pseudo-code for *KL*-cuts calculation.

For instance, in Figure 4.3 (a), starting by the backcut $\{u, v\}$ generated by the node s, the cuts $\{a, b, s\}$ from u and $\{s, g, h\}$ from v are combined, generating the incomplete *KL*-cut $\{\{a, b, s, g, h\}, \{u, v\}\}$. The last step adds the node r to the set of outputs, resulting in the complete *KL*-cut $\{\{a, b, s, g, h\}, \{r, u, v\}\}$ containing the nodes u, v, r and t.

To reduce the number of calculated *KL*-cuts, one can use only global cuts and global backcuts in the process, which produces *global KL-cuts*. They are fewer and larger *KL*-cuts. However, the covering of the circuit may get compromised. To ensure the covering, an additional round of *KL*-cuts generation could be done, this time using only local cuts and backcuts, creating *local KL-cuts*, and possibly only over the previously uncovered portion of the AIG. The collection of local and global *KL*-cuts together defines the *factor KL-cuts*.

A quantitative comparison between full *KL*-cuts enumeration and factor *KL*-cuts enumeration can be found in section 6.2.

As an example, let us consider the AIG shown in Figure 4.3. If the *KL*-cuts, with $K = 5$ and $L = 3$ (or simply 5-3-cuts), are computed based only on global cuts and backcuts one possible covering for the circuit is shown in Figure 4.3 (a), which is composed by the

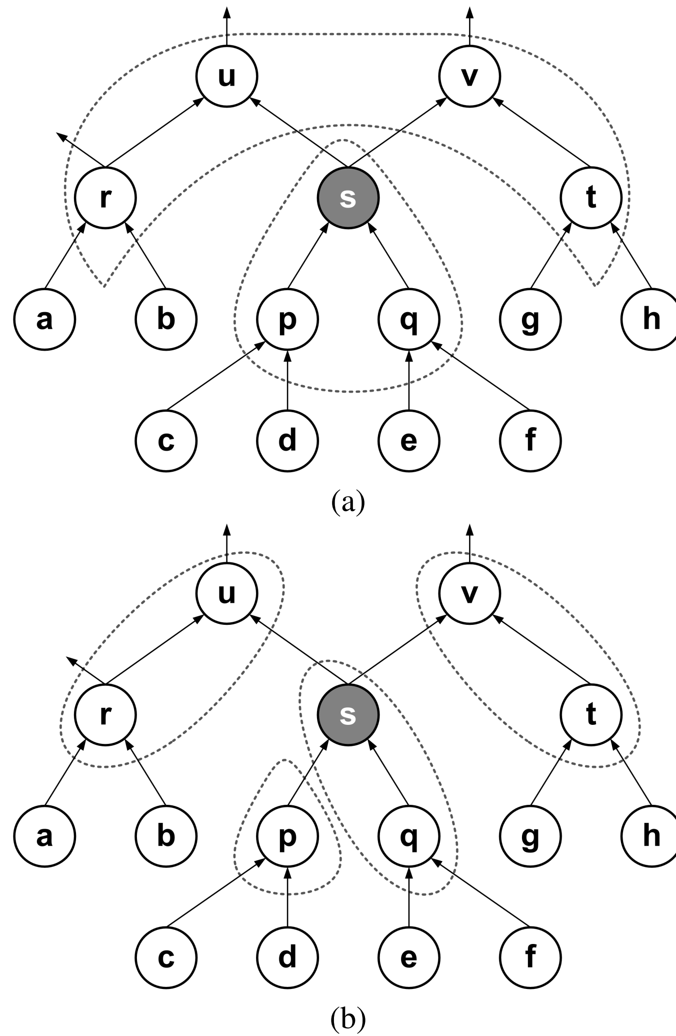

Figure 4.3: AIG illustrating covering. Nodes a, b, c, d, e, f, g and h are primary inputs. Nodes u and v are primary outputs. (a) A covering using 5-3-cuts. (b) A covering using 3-2-cuts.

KL-cuts $\{\{a, b, s, g, h\}, \{r, u, v\}\}$ and $\{\{c, d, e, f\}, \{s\}\}$. On the other hand, the circuit cannot be covered only by global 3-2-cuts. Under these conditions, only the nodes r, t, u and v can be covered. To complete the covering, local *KL*-cuts must be used, and a possible covering is shown in Figure 4.3 (b).

4.3 Unbounded KL-Cuts

Although the gains on scalability with *KL*-cuts come from the fact that the parameters K and L can be controlled, it can be also interesting to fix one of these values leaving the other unlimited. For that application, the *KL-cut with unbounded K* and the *KL-cut with unbounded L* are defined here.

Any *L*-feasible backcut has exactly one correspondent *KL*-cut with unbounded K, and that should be the largest *KL*-cut formed by that *L*-feasible backcut and a cut (not necessarily *K*-feasible). Similarly, any *K*-feasible cut has exactly one correspondent *KL*-cut with unbounded L, being that the largest *KL*-cut formed by a backcut (not necessarily L feasible) and the *K*-feasible cut in question.

4.3.1 KL-Cuts with unbounded K

The *KL*-cuts with unbounded K are very similar to regular *KL*-cuts, although they have no restriction on K, i.e. the number of inputs.

Figure 4.4 shows the pseudo-code for the algorithm. First of all, the *L*-feasible backcuts are calculated. Each backcut $\mathcal{G}_{\mathcal{L}}$ is the set of outputs of a *KL*-cut $\mathcal{G}_{\mathcal{KL}}$, so the set of inputs $\mathcal{G}_{\mathcal{K}}$ needs to be found. The graph is recursively traversed in depth first order (function ADDNODES()). Each node that neither is a PI, nor has a backcut composed exclusively by nodes on the set of outputs (function LCUTSOK()) is added to the set of inputs, otherwise the function is applied recursively to its child nodes. In other words, $\mathcal{G}_{\mathcal{KL}}$ only contains nodes n such that $\exists c \in \Phi_{\mathcal{L}}(n) | c \subseteq \mathcal{G}_{\mathcal{L}}$.

```
1:  function COMPUTEKLCUTSUNBOUNDEDK(aig, L)
2:      lcuts ← COMPUTELCUTS(aig, L)
3:      klcuts ← ∅
4:      for all lcut in lcuts do
5:          inputs ← ∅
6:          for all node in lcut do
7:              ADDNODES(node, inputs, lcut)
8:          end for
9:          klcuts.add(CREATEKLCUT(inputs, lcut))
10:     end for
11:     return klcuts
12: end function
13: function ADDNODES(node, inputs, lcut)
14:     if LCUTSOK(node, lcut) and node is not PI then
15:         ADDNODES(node.input(1), inputs)
16:         ADDNODES(node.input(2), inputs)
17:     else
18:         inputs.add(node)
19:     end if
20: end function
```

Figure 4.4: Pseudo-code for *KL*-cuts with unbounded K computation.

Notice that by applying this algorithm, any *L*-feasible backcut in the AIG leads to exactly one *KL*-cut with unbounded K, and this computation is performed in a single traversal of the graph. Observe also that the value of K is self adjusted by the topology and convergence of the graph.

Instead of computing all backcuts, it is possible to use only global backcuts, speeding up the process. Moreover, when using only global backcuts for calculating *KL*-cuts with unbounded K, the generated sub-graphs have factor trees as its elementary blocks. In other words, each one of these *KL*-cuts is constituted by one or more complete factor trees.

It is of particular interest the *KL*-cuts with unbounded K and $L = 1$, and based on global backcuts. In this case the sub-graph may contain more than one factor trees that are reconvergent to the sole output it presents. For instance, in Figure 4.5, the *KL*-cut with unbounded K for the global backcut $\{t\}$ is $\{\{a, b, c, d\}, \{t\}\}$. Observe the incorporation of 3 factor trees on this *KL*-cut. Moreover, there is only one possible covering of a circuit

by *KL*-cuts of this specific type, and its computation is done in a single traversal of the AIG. This performs a full partitioning of the circuit, and each partition will be an MFFC (CONG; DING, 1996). Some results are present in section 6.3.4.

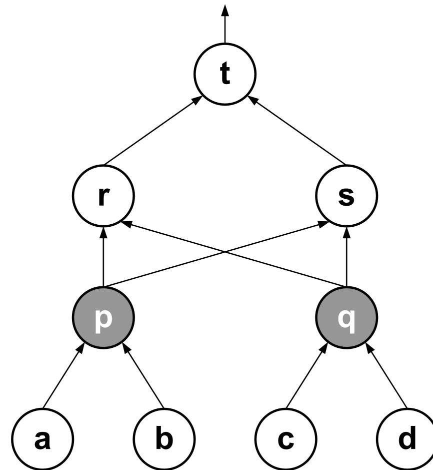

Figure 4.5: AIG exemplifying *KL*-cuts with unbounded K. The *KL*-cut $\{\{a, b, c, d\}, \{t\}\}$ is a *KL*-cut with unbounded K.

A *KL*-cut with unbounded K contains all nodes that are only used for producing its outputs. For instance, in Figure 4.1, the backcut $\{r\}$ leads to the *KL*-cut with unbounded K $\{\{a, p\}, \{r\}\}$, the backcut $\{s\}$ to $\{\{p, c, d\}, \{s\}\}$, and the backcut $\{r, s\}$ to $\{\{a, b, c, d\}, \{r, s\}\}$.

Observe that on this process no *K*-feasible cut needs to be calculated. Also, the use of global backcuts instead of all backcuts can reduce the total time.

4.3.2 KL-Cuts with unbounded L

The *KL*-cuts with unbounded L are *KL*-cuts with no restriction on L — the number of outputs. These *KL*-cuts contain all nodes that have a support defined by the starting *K*-feasible cut.

The pseudo-code for computing *KL*-cuts with unbounded L is described in figure 4.6. Initially, the *K*-feasible cuts are generated for all nodes. Then, for each cut $\mathcal{G}_{\mathcal{K}}$ a set of outputs $\mathcal{G}_{\mathcal{L}}$ needs to be generated, to define a *KL*-cut $\mathcal{G}_{\mathcal{KL}}$. The nodes are traversed from the starting cut nodes on the outputs direction (function ADDNODES()). Each node is tested by the function KCUTSOK(), which returns true if the node has at least one *K*-feasible cut formed only by nodes in kcut. That is, $\mathcal{G}_{\mathcal{KL}}$ contain nodes n such that $\exists c \in \Phi_{\mathcal{K}}(n) | c \subseteq \mathcal{G}_{\mathcal{K}}$.

To illustrate, considering figure 4.1, starting from the *K*-feasible cut $\{b, c, d\}$, the resulting *KL*-cut with unbounded L would be $\{\{b, c, d\}, \{p, s\}\}$.

On a technology mapping process the excessive number of outputs can be circumvented, a *KL*-cut with unbounded L having N outputs can be implemented as $\lceil N/L \rceil$ regular *KL*-cuts. As an example, a 5-5-cut can be made out of two 5-2-cuts and one 5-1-cut.

```
1:  function COMPUTEKLCUTSUNBOUNDEDL(aig, K)
2:      kcuts ← COMPUTEKCUTS(aig, K)
3:      klcuts ← ∅
4:      for all kcut in kcuts do
5:          outputs ← ∅
6:          for all node in kcut do
7:              ADDNODES(node, outputs)
8:          end for
9:          klcuts.add(CREATEKLCUT(kcut, outputs))
10:     end for
11:     return klcuts
12: end function
13: function ADDNODES(node, outputs)
14:     if KCUTSOK(node) and node is not PO then
15:         for all out in node.outputs do
16:             ADDNODES(out, outputs)
17:         end for
18:     else
19:         outputs.add(node)
20:     end if
21: end function
```

Figure 4.6: Pseudo-code for *KL*-cuts with unbounded L computation.

5 APPLICATIONS OF KL-CUTS

In this section possible applications for *KL*-cuts are discussed. Some of them were explored in this work, and results are shown in section 6. For other applications the discussion is qualitative, but no results were produced.

5.1 Technology Mapping

Multiple output cells are a reality on modern standard cell libraries, e.g. the full-adder and half-adder cells. Similarly, library free technology mappers could use multiple output cells to reduce the area of a circuit, especially on arithmetical circuits. Moreover, current FPGAs have multiple output LUTs available (HUTTON et al., 2004; COSOROABA; RIVOALLON, 2006). Hence the utilization of methodologies that only manage single output portions of circuits can lead to a poor quality result.

Technology mapping for FPGAs is somehow simpler than standard libraries. In a simplistic approach, every LUT on an FPGA is equal, so delay, area and power can be normalized to those of one LUT. Also, as the interconnections are already established on the circuit, there is no load variation, making the timing estimation much more reliable.

5.1.1 Greedy Covering

A simple algorithm to perform a full covering of a circuit by *KL*-cuts was elaborated. This algorithm does not intend to achieve the state-of-the-art in mapping, but to confirm the potential of using *KL*-cuts in technology mapping. The greedy algorithm searches for local maxima. At each iteration the largest possible *KL*-cut is chosen, and all *KL*-cuts with overlapping nodes are eliminated from the solution space. These iterations are repeated until the circuit is fully covered.

Two effort levels are defined. Using high effort level, all *KL*-cuts are present in the solution space of the algorithm, granting a search through the entire specter of possible cuts. When using low effort level, only global and local *KL*-cuts are available to the covering algorithm. This is a good heuristic because global cuts are the largest cuts available, but may not be sufficient to cover the circuit. For the areas where global cuts fail to cover, local cuts will be used.

Besides that, different flavors of this greedy covering were elaborated. The completely free covering, where the largest cut among all is chosen, is able to rapidly cover a large portion of the circuit, but leaving too many small portions uncovered and isolated, so when the time comes to cover these regions, only small cuts are available. Trying to address this deficiency, directed coverings are proposed. One is a top-down approach, where the only candidate cuts are the ones covering the topmost (output end) nodes of

the circuit. It can be seen as if successive layers are covered one by one. Alternatively a bottom-up covering can be used, which is similar to the top-down approach, but starting from the inputs. As most of the circuits are wider near the inputs than near the outputs, a bottom-up covering usually presents better results than a top-down approach.

As a proof-of-concept algorithm, it showed the usability of *KL*-cuts on a technology mapping algorithm on a multiple output flow. Results of some variations of this algorithm are shown in section 6.3.1.

5.1.2 Area Flow Covering for Multiple Outputs

The area minimization problem in the duplication free mapping can be solved optimally by decomposing the circuit into MFFCs, which are mapped for area. However with the use of controlled duplication, further area can be saved.

An extension to the algorithm of area flow covering (MANOHARARAJAH; BROWN; VRANESIC, 2006) is presented in this thesis, as the original one only deals with single output cuts. The area flow computation remains practically the same. The main differences are the classification of nodes, and the creation of different modes of operation.

Here, the idea of area flow of a node AF(n) is kept, but let us separate this into two concepts: area flow of a node AF(n) and area flow of a cut AF(c). If C_n is the set of cuts that have n as an output, then AF(n) is the smallest AF$(c_n)|c_n \in C_n$. In other words, the area flow of a node n is the smallest area flow between all cuts that generate n. This chosen cut c_n is called, as before, the *best cut* of n. The area flow of a cut is defined in equation 5.1. In this equation, $c_{\mathcal{K}}$ is the set of input nodes of c.

$$\text{AF}(c) = \frac{\text{Area}(c) + \sum_{n_i \in c_{\mathcal{K}}} \text{AF}(n_i)}{\text{Fanout}_{est}(c)} \tag{5.1}$$

The main change lies on how to compute Fanout$_{est}(c)$. Let us first define a classification system for nodes and cuts. In the original algorithm, nodes are classified only as visible or not visible. In this version the same node can have different classifications when regarding different cuts. Let us assume a given covering C. If $c_{\mathcal{L}}$ is the set of output nodes of a cut belonging to this covering $c \in C$, at least one $n \in c_{\mathcal{L}}$ is used, but not necessarily all of them. Even if a node n is effectively used, the cut c in question may not be the best cut of n, meaning that n is necessarily generated again by another cut.

In this context, for a node n and a cut c:

- If n has a fanout larger than zero and c is its best cut, then n is a *used* node in c.
- If n has a fanout larger than zero, but c is not its best cut, then n is a *virtually used* node in c.
- If n has a fanout zero, it is classified as a node *not used*, or not visible.

Notice that the fanout of a cut is the denominator in the area flow formula. Being so, it is of great impact on the choice of the covering, and should receive special attention.

Considering this scenario, the most coherent way of computing the fanout of a node, let us call it the *mode 1*, would be the one described in equation 5.2. Here, considering $c_{\mathcal{L}}$ as the nodes that are outputs of c, c_u denote the set of used nodes in $c_{\mathcal{L}}$, c_v the set of virtually used nodes in $c_{\mathcal{L}}$ and c_n the set of not used nodes in $c_{\mathcal{L}}$. Basically, the fanout of a cut is the sum of the fanout of its used nodes. If it does not have used nodes, then its fanout should be considered one.

$$\text{Fanout}(c) = \begin{cases} \sum_{n_i \in c_u} \text{Fanout}(n_i) & : c_u \neq \emptyset \\ 1 & : \text{otherwise} \end{cases} \tag{5.2}$$

Although this method produces a valid covering, it often fails on the extensive use of multiple output cuts. To overcome this limitation, two more method are proposed.

Mode 2 considers, as well as the used nodes, the virtually used nodes, as seen in equation 5.3. The idea of considering all visible nodes is that, if a multiple output cut was not the best cut at a previous iteration, it does not mean that it will not be at the next one.

$$\text{Fanout}(c) = \sum_{n_i \in c_u} \text{Fanout}(n_i) + \sum_{n_i \in c_v} \text{Fanout}(n_i) \tag{5.3}$$

Finally *mode 3* counts the fanout of every output, regardless if it is used or not, see equation 5.4. The inspiration for this is to use it at the first iterations, so the other methods will have as starting point a covering that uses plenty of multiple output cuts.

$$\text{Fanout}(c) = \sum_{n_i \in c_u} \text{Fanout}(n_i) + \sum_{n_i \in c_v} \text{Fanout}(n_i) + \sum_{n_i \in c_n} 1 \tag{5.4}$$

All of these modes of operation deal differently with nodes having different classifications. Nevertheless, if only single output *KL*-cuts are used, the three modes reduce to the original algorithm proposed for *K*-feasible cuts.

It is easily noticeable that the area flow when using modes 2 and 3 do not converge to the actual area used by the covering. This is not necessarily a problem, since the idea is to use them during the first iterations only, always finishing the covering using mode 1. Notice that the first iteration uses as fanout estimation the fanout of the nodes in the starting graph, so for this specific iteration the mode does not matter. An empirically found good distribution would be a covering achieved by the first iteration (in any mode), followed by two or three iterations on mode 3, the by two or three in mode 2, and finally by three to five in mode 1.

As an example table 5.1 shows the evolution of the area flow through the iterations by using only mode 1. Columns '# SO cuts' and '# MO cuts' divide the total number of cuts into single output and multiple output cuts respectively. This example is the same benchmark circuit as table 3.1. Even though the number of multiple output cuts was of the order of 20%, the reduction on the number of cuts when comparing to the single output version did not follow.

Now let us compare these results with table 5.2. This table shows the iterations of a mapping using, besides the first iteration, two using mode 3, two using mode 2 and five using mode 1, in this order. The first thing to notice is that, in modes 3 and 2, the reduction on the use of multiple output cuts is slower. Then, although the number of multiple output cuts in the end is slightly smaller, there is a reduction on the total number of cuts.

The authors of the original algorithm say that values of α between 1.5 and 2.5 produce good results. An α too small requires more iterations to let area flow to converge to the real area. On the other hand, an α too big may produce worst results or even make the area flow to oscillate and never converge.

As a brief example, consider the graph in figure 5.1. Consider that there are three cuts available to the covering: $c_1 = \{\{a, b, c\}, \{x, y, f, e\}\}$, $c_2 = \{\{a, b, c\}, \{d, f, z\}\}$ and $c_3 = \{\{a, b, c\}, \{x, y, z\}\}$. To make it clear, c_1 contains the nodes d, e, f, x and y, c_2 contains d, e, f and z, and c_3 all nodes except the primary inputs a, b and c. Consider also

Table 5.1: Effect of iterations in multiple output area flow, using only mode 1.

Iteration	# cuts	AF	# SO cuts	# MO cuts
1	3808	2700.78	2779	1029
2	3581	3047.68	2749	832
3	3548	3290.89	2785	763
4	3527	3414.07	2800	727
5	3515	3474.00	2806	709
6	3514	3498.49	2812	702
7	3512	3507.78	2812	700
8	3512	3510.44	2812	700
9	3512	3511.37	2812	700
10	3512	3511.77	2812	700

Table 5.2: Effect of iterations in multiple output area flow, using all modes of operation.

Iteration	# cuts	AF	# SO cuts	# MO cuts	Mode
1	3808	2700.78	2779	1029	*
2	3758	2799.64	2740	1018	3
3	3759	2864.14	2742	1017	
4	3598	3006.58	2748	850	2
5	3561	3139.40	2763	798	
6	3510	3285.54	2752	758	1
7	3493	3382.59	2764	729	
8	3468	3435.32	2777	691	
9	3464	3450.26	2785	679	
10	3465	3457.77	2787	678	

that this is a covering aiming a programmable device, so the area of every cut is unitary. In this scenario, table 5.3 contains values of $AF(c_i)$ in different situations. The column labeled “1st” contains the value of area flow on the first iteration, which is independent of mode. As a consequence of these values, the first covering would be constituted by cuts c_1 and c_2. In this moment, x and y are used nodes in c_1, and z is a used node in c_2. c_3 does not contain any used node, but x, y and z are virtually used in c_3. The other nodes are not used.

Table 5.3: Area flow according to different modes of computing fanout.

	1st	M1	M2	M3
c_1	$1/7$	$1/2$	$1/2$	$1/4$
c_2	$1/7$	1	1	$1/3$
c_3	$1/3$	1	$1/3$	$1/3$

Then three cases are considered. The columns labeled “M1”, “M2” and “M3” rep-

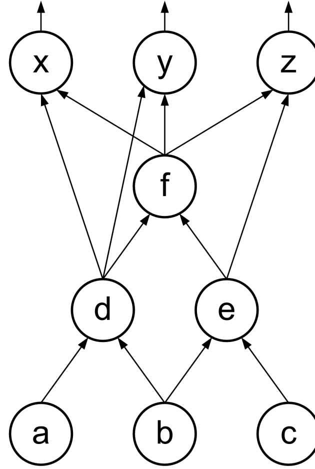

Figure 5.1: An AIG to illustrate the multiple output area flow algorithm.

resent area flow in modes 1, 2 and 3 respectively. In these columns equation 3.11 is bypassed by using $\alpha = \infty$, for the sake of an easy comprehension of this example. The best solution of this example would be a covering using only c_3, which would never be achieved by using only mode 1. Mode 2 leads to the best solution, since $AF(c_3)$ is the smaller area flow. In mode 3 cuts $AF(c_2) = AF(c_3)$, but this tie only occurs when looking to node z. Even if c_2 is effectively chosen by z, as the final passes are always in mode 1, c_2 would be dropped in favor of c_3. So, although the other modes are not directly related to the area, they contribute to a better result by promoting the use of multiple output cuts.

Although the AIG representation is loop free, an additional precaution must be taken to avoid loops at a *KL*-cuts abstraction level. Figure 5.2 shows a loop free AIG that have loops when looking at *KL*-cuts as black boxes. The AIG is divided in two *KL*-cuts: $c_1 = \{\{a, b, g\}, \{e, f\}\}$ and $c_2 = \{\{c, d, e\}, \{g, h\}\}$. The cut c_1 has as input the node g, which is generated by c_2, and the cut c_2 has as input the node e, which is generated by c_1. This defines a loop, since c_1 depends on c_2 which depends on c_1. As the algorithm runs the graph depth-first at a *KL*-cuts abstraction level, it must be loop free. One simple way of avoiding these loops is explained as follows.

When looking a *KL*-cut $\mathcal{G}_{\mathcal{KL}}$ isolated, consider the height of a node $h(n)$ as the largest number of hops from the node n to a node $n_{in} \in \mathcal{G}_{\mathcal{K}}$. One necessary condition for a loop to exist is that $\exists n_{in} \in \mathcal{G}_{\mathcal{K}}, \exists n_{out} \in \mathcal{G}_{\mathcal{L}} | h(n_{in}) \geq h(n_{out})$. For example, looking at the cut $c_1 = \{\{a, b, g\}, \{e, f\}\}$, $h(g) = h(e)$, being g an input of c_1 and e an output. The simplest way to avoid having loops is to exclude from the solution space every cut that does not respect this height relation restriction.

Another consideration to be made is about the way *KL*-cuts are constructed. In the generation algorithm, a final step assures that every node pointing to another one outside the cut is listed as an output. As an example, in figure 5.2, the incomplete cut $\{\{a, b, g\}, \{f\}\}$ would be completed to form the cut $\{\{a, b, g\}, \{e, f\}\}$. This is vital for an application such as AIG rewriting or other peephole optimization procedure (see section 5.3), because otherwise a signal could be ignored and be not generated. But for this application an incomplete *KL*-cut causes no harm, and a proof could be derived from the fact that that any *K*-feasible cut can be seen as an incomplete *KL*-cut. It is worth to point out that the proposed multiple output area flow covering algorithm, when fed with *K*-feasible cuts represented as incomplete *KL*-cuts, behaves exactly as the original single

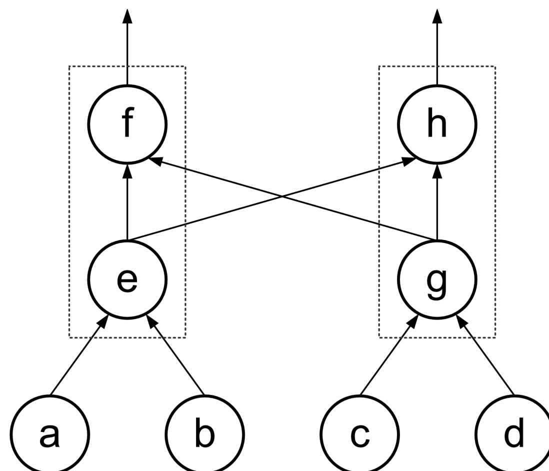

Figure 5.2: An AIG with a loop formed by *KL*-cuts.

output area flow algorithm, and thus can be considered a generalization of it.

One final step, after concluded the covering process, removes the outputs that are not used in the covering, possibly turning some cuts incomplete, and also possibly excluding some cuts form covering, in case all of its outputs were removed. This reduces the task of cells generation, if a library-free mapping is the target. This step may also be excluded if a library based approach is being followed.

Finally it must be highlighted that this method is still under investigation. Different modes of operation and different modes distributions throughout the mapping process are still to be tested. Nevertheless, results of the current state of the algorithm are shown in section 6.3.2.

5.1.3 Matching

The covering process needs, besides the structure to be covered, a search space formed by cells. This search space, or solution space, is composed by the matches between portions of the subject graph and a predefined library.

When mapping using a library-based approach, the matching phase consists in deciding whether the sub-circuit has a correspondent cell in the library. In this context, “correspondent” may have several definitions. On a structural matching, a sub-circuit matches a cell if they have an isomorphic representation. It is possible that the same library cell is associated to more than one graph decomposition. On a Boolean matching, a match is when the Boolean function implemented by the sub-circuit belongs to the same equivalence class as the cell function.

On a library-free mapping, the solution space is constituted by the cuts that respect a determined restriction. If it is a matter of number of inputs and outputs, then the generation already takes care of generating only useful cuts. On the other hand, if the restriction is more sophisticated, like the number of series transistors, then an algorithm must be run on each cut to determine if it must be pruned out of the solution space before the covering process.

Even in the case of a library-free mapping, if the target is not a programmable device, a matching algorithm is required. The result of the covering phase is a set of cuts. Place and route algorithms, which is usually the following step on a design flow, requires each cell to have a layout associated. In a library-free flow this layout generation step is postponed to after the covering is complete. Being so, a grouping of the cells chosen by the technology

mapping into equivalence classes is necessary, so only one layout for each equivalence class can be generated instead of one layout per cell.

As the Boolean matching methods are more comprehensive then structural matching methods, an extension to the TEMPLATE method (HINSBERGER; KOLLA, 1998) is proposed in this thesis in order to contemplate cells having multiple outputs.

5.1.3.1 A Multiple Output Boolean Matching Method

As well as the concepts of P, NP and NPN-equivalence classes are defined for a single function, the ideas of PP, NPP or NPNP-equivalence classes can be defined for a list of functions, which can be viewed as a single multiple output function $f : B^n \mapsto B^m$.

A PP-equivalence class is defined by allowing permutations both in inputs and outputs. As an example, consider the following lists of functions:

$$s_1 = \begin{cases} f_1 = a * b + c \\ f_2 = b * c \end{cases}$$

$$s_2 = \begin{cases} f_1 = a * c \\ f_2 = a * c + b \end{cases}$$

$$s_3 = \begin{cases} f_1 = a * c \\ f_2 = a * b + c \end{cases}$$

Here s_1 belongs to the same PP-equivalence class as s_3. Starting from s_3, it suffice to switch the inputs a and b, and the outputs f_1 and f_2 to obtain exactly s_1. Now comparing s_1 and s_2: although f_1 of s_1 (let us call it $s_1.f_1$ for a simpler notation) is P-equivalent to $s_2.f_2$, and $s_1.f_2$ is P-equivalent to $s_2.f_1$ — a condition necessary to P-equivalence, but not sufficient —, s_1 is not PP-equivalent to s_2. This is because its internal functions are only P-equivalent to each other in different input permutations, and there is no permutation that makes both $s_1.f_1$ P-equivalent to $s_2.f_2$ and $s_1.f_2$ P-equivalent to $s_2.f_1$.

Similarly an NPP-equivalence class is defined by allowing input negation and permutation, and output permutation. And an NPNP-equivalence class is defined by allowing inputs and outputs to be both negated and permuted. Two lists of functions with a different number of inputs or a different number of outputs never belong to the same equivalence class. This work is focused in PP-equivalence class matching.

This thesis proposes an extension of the TEMPLATE system (reviewed in section 3.5), in order to be able to find PP-equivalence between lists of functions. The focus application is to determine multiple output cells equivalence. The idea is similar to the original algorithm, to find a representative list of functions $R[L_f]_{PP}$ of $[L_f]_{PP}$.

The truth table representation of a list of functions (figure 5.3 shows a truth table for a list L_1) has more than one column of outputs, so its representation as a bit string is not trivial. For this algorithm, the bit sequence is composed by interspersing the bit string representations of each function. As an example, the bit string representation of L_1, of figure 5.3, is 0011010001100011011100110. This representation is changed whenever the input ordering or the output ordering is changed. Once the lists of functions can be represented this way, they can be compared using the unsigned integer semantic, thus establishing a relation of order. This way, the representative function is defined as the maximum list in the same equivalence class, as shown in equation 5.5.

$$R[L_f] = \max_{L_i \in [L_f]} L_i \tag{5.5}$$

c	b	a	f_1	f_2	f_3
0	0	0	0	0	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	1
1	1	0	1	0	0
1	1	1	1	1	0

Figure 5.3: A truth table representation of a set of functions L_1.

The bit string is defined this way in order to keep profiting from the property shown in figure 3.4, discussed in section 3.5. If the order of variables is defined up to the k-th variable, then the first 2^k lines of the truth tables are defined. Consequently, the first 2^k times the number of functions bits of the bit string representation are also defined. So, by defining this rule of bit string, the non-maximal branches are pruned, just like in the original algorithm.

Symmetry information can also be used, although potentially with less profit than in the single output version. In order to be able to cut off a branch of the search space, the variables must be symmetrical in all functions. The probability of this to happen decreases as the number of outputs grow. So, in the current implementation this feature is not present.

Figure 5.4 shows a tree denoting the input variable permutations tested. At each node, the already defined bits are shown interspersed in the output order that produces the maximal integer. The output order already defined is also displayed. If some outputs still do not have an established order, they are grouped together.

Following the example, at the starting node $()$ no variable ordering is defined. This way only the first line of the truth table is known. Two of the three functions evaluate to 0, and f_3 to 1. Being so, the bit string can start in three different ways: 001, 010 or 100. As 100 is larger than the others, it is the chosen one and determines that f_3 is the first function. The function ordering (or output ordering) is represented immediately below the bit string, as $\{\{3\},\{1,2\}\}$, meaning that first comes output 3, followed by, in unknown relative order, outputs 1 and 2.

Each child node adds one variable ordering regarding its father. For example, in node $\binom{1}{1}$ the first position is occupied with the first variable (a). Node $\binom{1}{2}$ attributes to the first position variable 2 (b), and so on. As the tree is constructed in breadth-first order, each time a level is completed, all non-maximal nodes can be abandoned. In the example, node $\binom{1}{1}$ is larger than the others, so the son nodes of $\binom{1}{2}$ and $\binom{1}{3}$ are never reached.

In the end, all leaves reached will have the same integer representation. If the output ordering is not completely specified in a leaf, then any order contained in the order specification may be used.

Results demonstrating the performance of this algorithm is shown in section 6.3.3.

By having the representative list of functions already computed, the matching reduces to a single equality comparison. If $R[L_1]_{PP} = R[L_2]_{PP}$, then they are PP-equivalent.

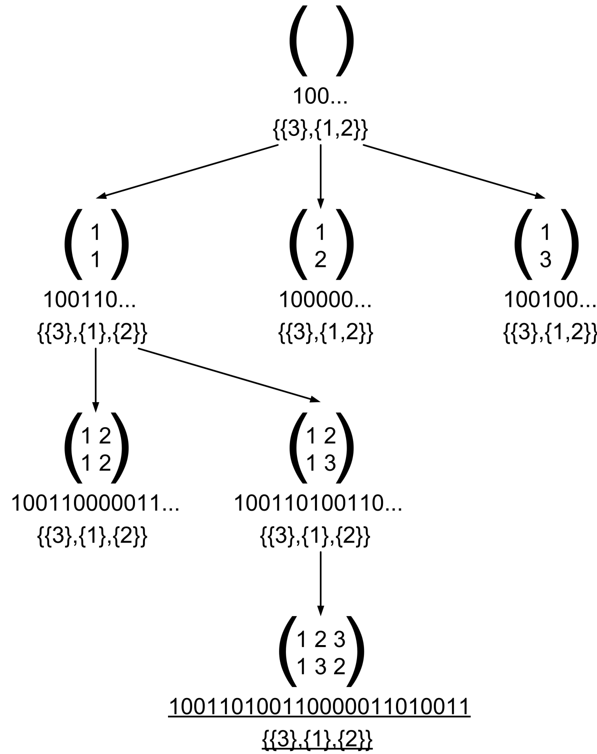

Figure 5.4: Computing $R[L_1]_{PP}$.

5.1.4 Partitioning

Technology mapping algorithms often make use of heuristics in order to reduce the complexity of the problem. One of the most popular heuristics is the partitioning.

Partitioning consists into breaking the circuit to be covered into several smaller parts, performing the mapping individually on each part, unifying the mapping in the end. It can make use of a post-processing step to correct mispredictions or to improve some characteristic. The decomposition of a graph in a forest of trees is an example of partitioning.

A decomposition that is a little more comprehensive than breaking into trees is breaking the circuit into MFFCs. A fanout-free cone of a node v FFC$_v$ is a cone which every node has its fanout nodes inside the cone. For each node v there is a unique maximum fanout-free cone MFFC$_v$ that contains every FFC$_v$ (CONG; WU; DING, 1999).

A *KL*-cut with unbounded K and $L = 1$ is an MFFC, and there is exactly one possible covering of a circuit into MFFCs. Hence, *KL*-cuts provide a way to decompose circuits into MFFCs. More than that, cuts with unbounded K may also be used with larger L to perform a partitioning. These cuts may be larger than MFFCs, but small enough to be able to be treated by some exhaustive algorithm. Further discussion and some results are shown in section 6.3.4.

5.2 Regularity Extraction

Implementing logic blocks in an integrated circuit in terms of repeating regular geometry patterns can provide significant advantages in terms of manufacturability and design cost (KHETERPAL et al., 2005). Regularity extraction, which means to detect recurring functionalities during logic synthesis, can constrain the physical design phase to exploit the regular netlist produced, going towards a DFM approach (ROSIELLO et al., 2007).

Design for Manufacturing (DFM), can be defined as a set of techniques adopted to estimate, control, and improve the yield and robustness of a circuit prior to fabrication.

One possible way of profiting from the use of *KL*-cuts in regularity extraction is explained step-by-step as follows:

- Enumeration of *KL*-cuts over the circuit structure;
- Pruning of the cuts by a predefined rules system, either restrictions from a virtual library, or even a heuristic;
- Grouping the cuts using a matching engine, either using a structural matching (faster) or a Boolean matching (more comprehensive);
- Attributing costs to these potential cells, by considering, besides electrical and physical properties, the number of matches each equivalence class, or template, has;
- Performing a mapping using this cost system.

This way the regularity of a mapping can be favored in a technology binding process.

5.3 Peephole Optimization

Another application for *KL*-cuts is to perform peephole optimizations. By defining a sub-graph on an AIG, it can be replaced by any other sub-graph that implements the same Boolean functions, but minimizing a given cost function. An example of peephole optimization is AIG rewriting (MISHCHENKO; CHATTERJEE; BRAYTON, 2006), explained in section 3.2.

This can be interesting also in a post-mapping stage (WERBER; RAUTENBACH; SZEGEDY, 2007), when it is usually called In-Place Optimization (IPO). Here several sub-circuits are analyzed and the ones that could produce improvement to some characteristic, typically area or delay, are selected. This sub-circuits are then replaced by a single complex cell, generated on-the-fly, and the resulting circuit is reanalyzed in order to evaluate improvements.

A variation of this IPO using *KL*-cuts, which was preliminarily studied, is described as follows. After the circuit is mapped on a standard cell library, *KL*-cuts are enumerated over the circuit, but only allowing cuts that do not break the current cells. In other words, the elementary block of these cuts are not the nodes of the AIG, but the cells on the previous mapping. Then, the cuts are grouped in PP-equivalence classes, and the most frequent functionalities are enumerated. So the automatic cell generator can create a complex cell by compacting these cells into one single cell, trying to reduce area. Finally, these cells replace the cuts, and the circuit is evaluated again for measuring improvements.

6 RESULTS

The results obtained come from implementations in Java language executing on a 2.4GHz Intel Pentium IV with 2GB of RAM. The benchmark circuits used are the largest circuits from ISCAS'85 and ISCAS'89. The AIGs were generated from original BLIF files by using the ABC tool (Berkeley Logic Synthesis and Verification Group, 2010). ABC performs a structural hashing (MISHCHENKO; CHATTERJEE; BRAYTON, 2005) in order to construct the AIG, and after that the command '*dc2 -l*' is executed twice in order to minimize node count, resulting on an efficient AIG representation.

Table 6.1 shows the profile of the circuits that compose the benchmark set. The number of nodes range from 400 to 11000 nodes, and the concentration of dag nodes varies from 20% to 40%. As for the sequential circuits, the flip-flops were stripped off, being replaced for an input/output pair, therefore only the combinational part remains.

Table 6.1: Benchmark information.

Name	Nodes	% Dag
C1355	424	38.68
C1908	385	40.78
C2670	680	19.71
C3540	947	24.29
C5315	1467	23.86
C6288	1902	73.82
C7552	1526	39.78
s13207	2849	23.20
s15850	3439	27.25
s35932	10837	30.96
s38417	9872	24.57
s38584	11554	25.64
Avg.	3824	32.71

6.1 L-Feasible Backcuts

Table 6.2 shows a brief comparison between complete L feasible backcuts enumeration and factor backcuts enumeration for $L = 2$. Although on the average, factor backcuts represent almost as many cuts as the complete enumeration and take a little longer to compute, for some circuits it represents much lesser cuts. This depends essentially on

the topology of the circuit, being hard to determine beforehand. The gains of using factor backcuts become more evident as the value of L grows. For example, for $L = 4$ the factor cuts sum a half of the complete enumeration, and are computed in 75% of the time. But for now the applications are limited to small values of L.

Table 6.2: Comparison between *L*-backcut enumeration and factor *L*-backcut enumeration.

	All		Factor			
Name	Total	Time (s)	Global	Time(s)	Local	Time(s)
C1355	1707	0.06	1081	0.02	915	0.02
C1908	1805	0.03	997	0.02	966	0.02
C2670	4342	0.06	1285	0.03	2564	0.03
C3540	6986	0.13	1361	0.03	6176	0.09
C5315	9960	0.16	3248	0.08	4787	0.08
C6288	7682	0.16	5871	0.16	4638	0.13
C7552	16790	0.30	4452	0.13	5037	0.16
s13207	10796	0.33	3233	0.17	10405	0.22
s15850	13715	0.39	4873	0.34	10490	0.34
s35932	90147	3.91	23484	2.91	31937	2.94
s38417	40936	2.38	14011	2.30	28767	2.48
s38584	39643	3.53	13969	3.08	33907	3.20
Avg.	20376	0.95	6489	0.77	11716	0.81

6.2 KL-Cuts

Figure 6.1 shows the number of *KL*-cuts enumerated on the largest circuit of the benchmark set, circuit *s38584*, varying K for some L values. Figure 6.2 shows the time taken to compute these cuts. It is visible that both the number of cuts and the time taken to compute them grow exponentially with K. Following, figures 6.3 and 6.4 present the same results, but changing the x-axis from K to L. This view of the same data reveals that the growth of both the number of cuts and the time taken to compute them is also exponential in relation to L. In figure 6.4 it is visible a certain discontinuity when changing L from 2 to 3. This is because, as said in section 4.2, if $L = 2$ there is a trick to avoid checking the cuts for connectivity. If $L = 1$ the cuts will always be connected, no test being needed. So, this step is the time taken on this test. This phenomenon can also be viewed in figure 6.2, as the lines for $L = 3$ and $L = 4$ are have a larger slope than $L = 1$ and $L = 2$. Part of it is because for larger L the graphs are larger, and the connectivity test takes more time.

Table 6.3 shows the results for constructing all *KL*-cuts and for factor *KL*-cuts enumeration. For 5-2-cuts, factored enumeration produced 56% of the number of *KL*-cuts generated from full enumeration, taking about the same run-time. If only global cuts are considered, then the process is done in half of the time. Global cuts can be used as a heuristic in the covering process, being local cuts used only in the portions of the circuit where global cuts fail to cover. It is also noticeable that global *KL*-cuts have more nodes on average than *KL*-cuts produced by complete enumeration, which is shown in the

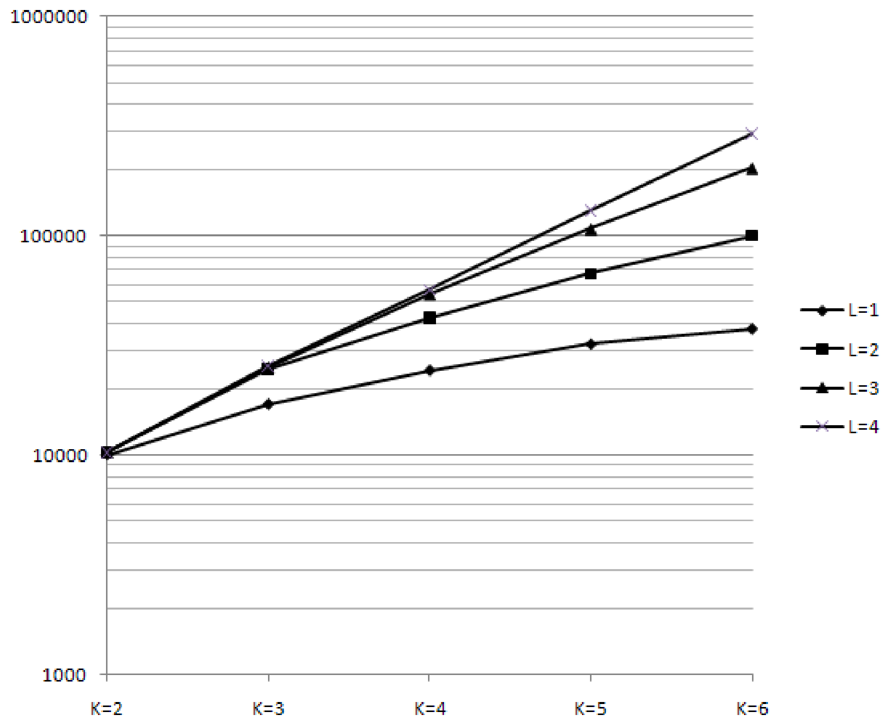

Figure 6.1: Number of *KL*-cuts versus K.

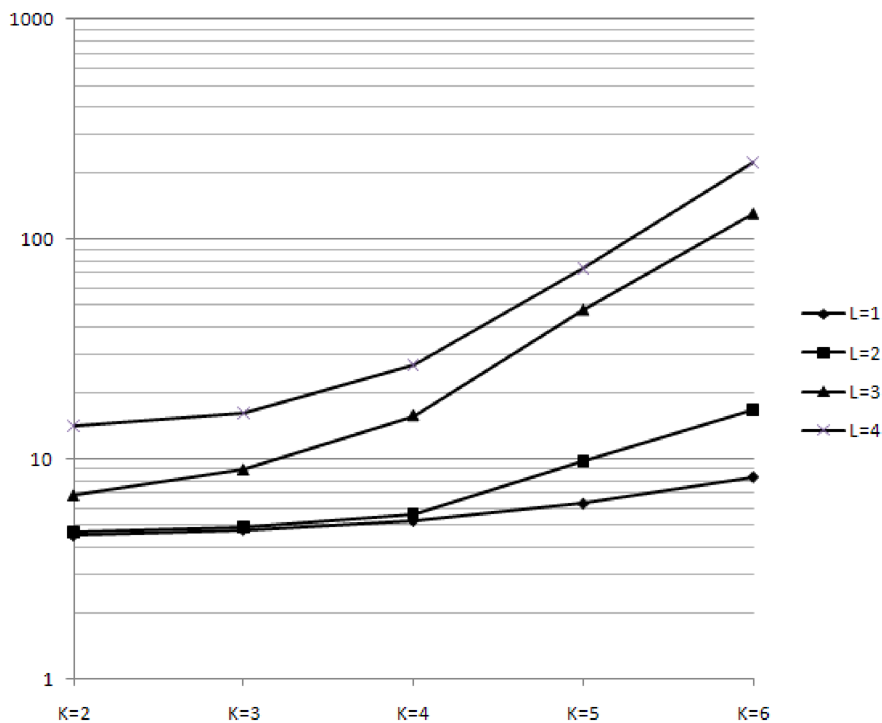

Figure 6.2: Time taken to compute *KL*-cuts versus K.

columns labeled *Size*.

6.3 Covering Algoritms

Modern FPGAs are capable of implementing LUTs with two outputs (HUTTON et al., 2004; COSOROABA; RIVOALLON, 2006). They consist essentially of two LUTs

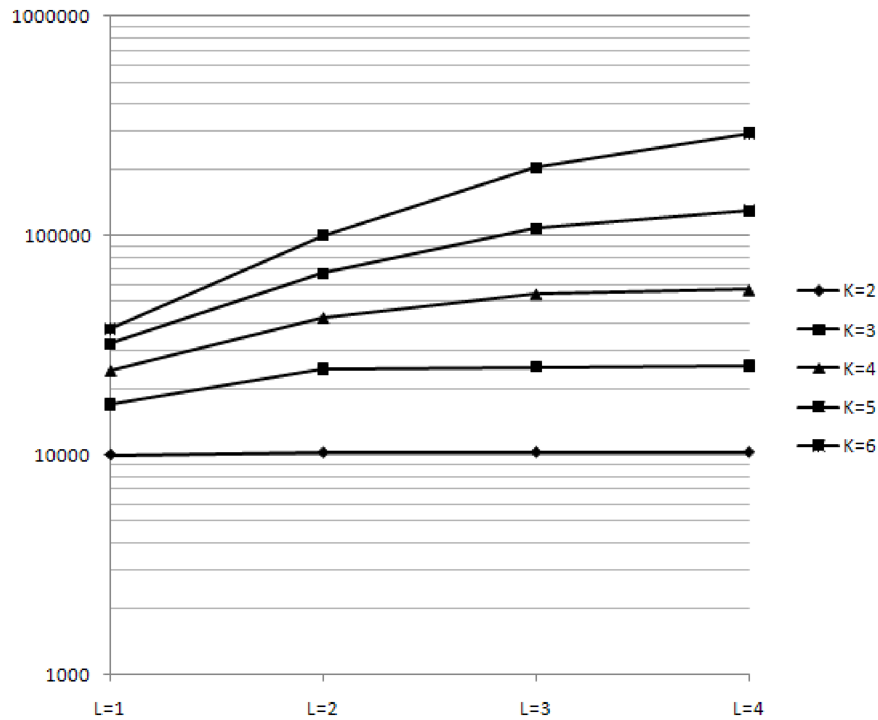

Figure 6.3: Number of *KL*-cuts versus L.

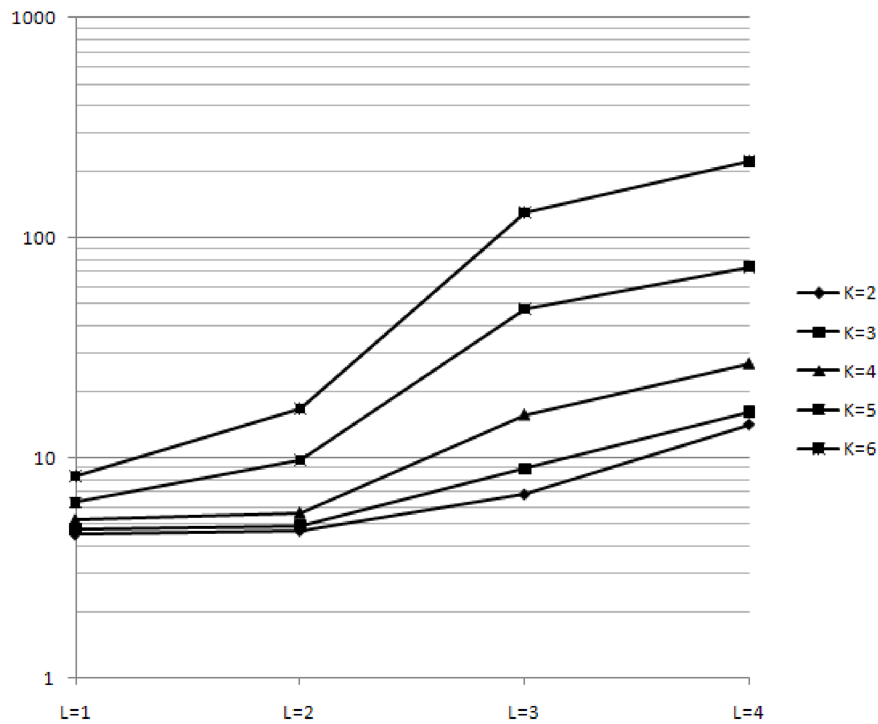

Figure 6.4: Time taken to compute *KL*-cuts versus L.

packed together. The main advantage of doing so is to alleviate the congestions on routing, although it produces some gain area. Generally a multiple output LUT can work on different configurations, for example some work either as a 6-1 LUT or as a 5-2 LUT. As each vendor and each model has specific configurations, it is assumed here that all LUTs are multiple output capable, but it is not necessary that all outputs are effectively used in a mapping.

Table 6.3: Comparison between *KL*-cuts enumeration and factor *KL*-cuts enumeration.

	All			Global			Local		
Name	Total	Size	Time	Total	Size	Time	Total	Size	Time
C1355	4646	6.11	0.55	1228	7.67	0.11	670	1.77	0.05
C1908	3344	5.00	0.38	790	5.69	0.09	806	2.28	0.05
C2670	5920	4.34	0.42	293	5.80	0.08	2057	2.79	0.11
C3540	9458	3.47	1.03	520	3.32	0.13	4632	2.91	0.31
C5315	12863	4.19	1.30	818	4.91	0.28	5254	2.76	0.25
C6288	11727	3.59	4.31	4787	3.79	2.11	3738	1.63	0.34
C7552	15283	4.67	2.45	1904	4.24	0.42	3909	2.48	0.44
s13207	12150	3.10	1.34	1275	2.94	0.48	8073	2.77	0.66
s15850	20821	3.50	2.13	2072	3.44	0.72	9928	2.70	1.03
s35932	89065	5.11	11.94	8740	5.56	6.27	36012	3.05	6.88
s38417	69215	3.43	9.28	3818	3.47	4.56	32691	2.80	5.83
s38584	67163	3.24	9.97	5528	3.16	6.81	40231	2.86	7.39
Avg.	26805	4.15	3.76	2648	4.50	1.84	12333	2.57	1.94

Table 6.4 shows some results on single output mapping, which will serve as a comparison to the multiple output algorithms. Two methods were used. The first, shown in column “ABC” shows the number of LUTs used by mapping the circuit running the ‘ *st; dch; if -C 12; mfs -W 4 -M 5000*’ commands (MISHCHENKO et al., 2009) on ABC four times, and picking the best result, with a library containing LUTs with up to 5 inputs. This library assumes the same cost for each LUT. Constants, buffers and inverters are not considered in this value. The second method is the area flow method (MANOHARARAJAH; BROWN; VRANESIC, 2006), reviewed in section 3.6, after eight iterations and using $\alpha = 2$. It can be noticed that area flow is almost three times faster than ABC, but produces slightly worse results.

6.3.1 Greedy Covering

Tables 6.5 and 6.6 show results for using all cuts and only factor cuts respectively, on a greedy mapping using a bottom-up (from inputs to outputs) approach. Columns named “Cuts” show the number of cuts for the resulting covering. Columns “% MOC” are the percentage of cuts that have multiple outputs. Two single output cuts can be implemented by a single multiple output LUT, as long as the sum of the number of inputs of these cuts is no larger than the LUT number of inputs. For example, a 2-1-cut can always be combined with a 3-1-cut, if we have a 5-2-LUT available. For that reason the columns labeled “LUTs”, which show the number of LUTs necessary to implement the covering, present a lower value than the columns “Cuts”. The columns “% MOL” represents the percentage of used LUTs that uses both outputs.

The conversion between the number of cuts and the number of LUTs, in this context, depends only on the number of inputs of each cut, and not from the actual inputs. It is a matter of counting how many single output cuts can be combined with each other. On a typical multiple output flow, all the covering process is performed on a single output basis, and finally a series of comparisons are made throughout the inputs of cuts, trying to bind together the ones with shared inputs. This comparison process can be very ex-

Table 6.4: Covering for single output LUTs using ABC and Area Flow methods.

	ABC		Area Flow	
Name	LUTs	Time (s)	LUTs	Time(s)
C1355	68	4.11	66	1.27
C1908	105	4.12	101	0.92
C2670	149	2.91	127	1.08
C3540	276	15.30	285	2.13
C5315	324	10.69	339	2.73
C6288	501	51.16	711	7.11
C7552	372	14.05	373	4.48
s13207	718	12.69	707	2.89
s15850	966	16.83	945	4.86
s35932	2493	14.95	2682	16.34
s38417	2659	49.22	2648	17.89
s38584	2655	29.27	2754	19.08
Avg.	941	18.78	978	6.73

pensive, and it is exactly this computation that is avoided by using our approach, which considers multiple output blocks from the beginning. These blocks with shared inputs will automatically be found as multiple output blocks, and ideally no single output block would have shared inputs to be combined (or else they would have been found).

Table 6.5: Greedy bottom-up covering using *KL*-cuts.

Name	Time (s)	Cuts	% MOC	LUTs	% MOL
C1355	0.95	74	37.84	54	88.89
C1908	0.72	75	69.33	64	98.44
C2670	1.16	141	35.46	110	73.64
C3540	2.38	273	47.25	223	80.27
C5315	3.42	326	44.48	269	75.09
C6288	5.06	359	91.36	344	99.71
C7552	3.70	333	66.07	285	94.04
s13207	5.20	589	55.69	487	88.30
s15850	8.33	819	47.86	657	84.32
s35932	87.69	2221	55.74	1746	98.11
s38417	70.84	2159	61.32	1809	92.54
s38584	112.66	2631	40.94	2123	74.66
Avg.	25.18	833	54.45	681	87.33

By comparing tables 6.5 and 6.6, it is clear that it is a matter of trading off the quality of the results and speed. The restriction on the use of only factor cuts drops the execution time to less than a half, but results on 10% more cuts and 5% more LUTs. The reason of this difference on the reduction of cuts and LUTs is because the covering with factor cuts, on the regions where local cuts are used, is performed with cuts with lesser inputs, hence

Table 6.6: Greedy bottom-up covering using only factor *KL*-cuts.

Name	Time (s)	Cuts	% MOC	LUTs	% MOL
C1355	0.56	75	33.33	55	81.82
C1908	0.52	81	56.79	69	84.06
C2670	0.78	152	11.18	114	48.25
C3540	1.13	304	18.42	243	48.15
C5315	1.89	358	18.72	282	50.71
C6288	3.63	359	93.59	348	99.71
C7552	1.67	341	53.37	289	80.97
s13207	2.92	662	31.27	513	69.40
s15850	4.30	859	27.82	664	65.36
s35932	34.25	2478	29.06	1805	77.17
s38417	23.48	2564	16.97	1947	54.03
s38584	40.44	2712	26.62	2224	54.41
Avg.	9.63	912	34.76	713	67.84

the ones that are single output are more easily “combinable” to produce a single LUT.

It was also implemented a undirected version of the covering, where the absolute largest cuts are chosen regardless their position on the graph. The average number of cuts and LUTs was practically the same (less than 1% of difference), but the execution time was, on average, 7 times larger. This shows that the use of a bottom-up approach is an efficient heuristic on this kind of covering.

The algorithms find a good fraction of *KL*-cuts which are naturally multiple output, and most of the single output *KL*-cuts found have few inputs, which allows its combination leading to a high utilization of multiple output LUTs (more than 85% on average when using all cuts), resulting on fewer LUTs used on the mapping (30% reduction comparing to ABC mapping).

Although this greedy covering has produced results with fewer LUTs than ABC mapping, it does not mean that it is a better algorithm. It must be taken in consideration that the LUTs used by ABC have one output, whereas our LUTs have two. So, ideally, our mapping should produce a 50% reduction on the use of LUTs, if the mapping has equivalent quality, and if the topology of the circuit allows such a reduction. On the other hand, if even with such a simplistic covering algorithm the results were relatively good, it means that this strategy of treating multiple output blocks from the beginning is on the right direction.

6.3.2 Area Flow Covering for Multiple Output

The area flow covering for multiple outputs algorithm, proposed in section 5.1.2, produced the results shown in table 6.7. The first thing to notice is that, even with the use of artifices like the different modes of computing fanout, the utilization of multiple output cuts is much smaller than produced by the greedy covering. Also, the algorithm tends to select cuts with a large number of inputs, making difficult the combination of single output cuts into a multiple output LUT. On the other hand, even with this sub-utilization of multiple output LUTs, the total number of LUTs is less than 10% larger than the greedy covering. In other words, there is more room for improvement in this algorithm than in

the greedy covering algorithm.

Table 6.7: Covering using the area flow for multiple outputs algorithm.

Name	Time (s)	Cuts	% MOC	LUTs	% MOL
C1355	1.81	71	9.86	64	21.88
C1908	1.44	83	24.10	66	56.06
C2670	1.59	119	9.24	114	14.04
C3540	3.08	252	11.11	235	19.15
C5315	4.20	314	7.32	302	11.59
C6288	15.39	241	98.76	240	99.58
C7552	6.77	302	19.87	298	21.48
s13207	4.09	645	9.61	564	25.35
s15850	6.42	850	10.94	736	28.13
s35932	28.88	2266	23.74	1664	68.51
s38417	25.61	2401	7.71	2168	19.28
s38584	25.84	2501	9.96	2364	16.33
Avg.	10.43	837	20.18	735	33.45

One particular circuit had utilization of almost 100% of multiple outputs cuts, where the number of LUTs used was a half of the used by ABC. The *c6288* circuit is a highly regular circuit, which implements a 16 by 16 bits multiplier in an array of adders.

Another advantage of this area flow algorithm over the greedy covering is explained as follows. The original area flow algorithm is able to find the depth minimum covering, and then perform area recovery on the following steps. This multiple output version could have this characteristic as well, but the greedy covering cannot.

6.3.3 Multiple Output Matching

The graphic of figure 6.5 shows the performance of the developed algorithm. The number of inputs varied from 4 to 16, and the number of outputs from 1 to 4. Each point in the graph is an average value obtained after 100 executions over randomly generated functions. It is noticeable that the dependency of the time consumed on the number of inputs is higher than an exponential approximately up to $K = 11$, and then it becomes a line in the semi-log graphic, denoting an exponential. The line representing a single output cut is approximately the time taken by the original algorithm, but without the symmetry optimization. Figure 6.6 shows the same results, but having L instead of K in the x-axis. In this graph it can be seen that the execution time has also an exponential relation with the number of outputs. However the slope is much less steep, meaning that is cheaper to add an output than adding an input.

6.3.4 Partitioning

A comparison between the circuit covering by factor trees and by *KL*-cuts with unbounded K and $L = 1$ is given in Table 6.8. The column labeled “Total” shows the number of sub-graphs necessary to perform the covering, the column “Size” shows the average number of nodes per sub-graph, and the column “Mean K” shows the average number of inputs of the sub-graphs. Notice that the increase in the number of nodes

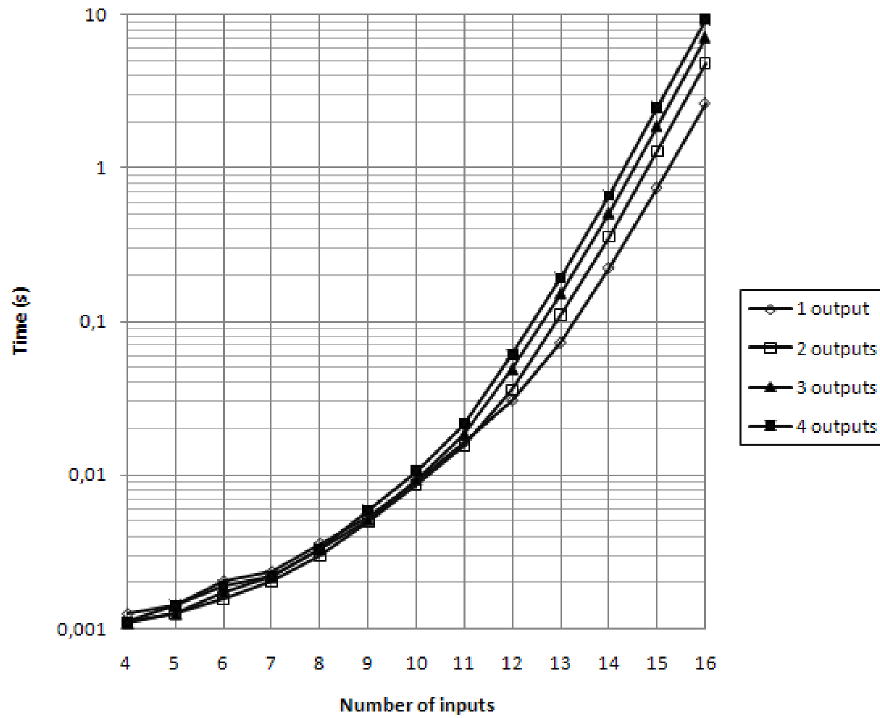

Figure 6.5: Execution time of matching algorithm varying the number of inputs.

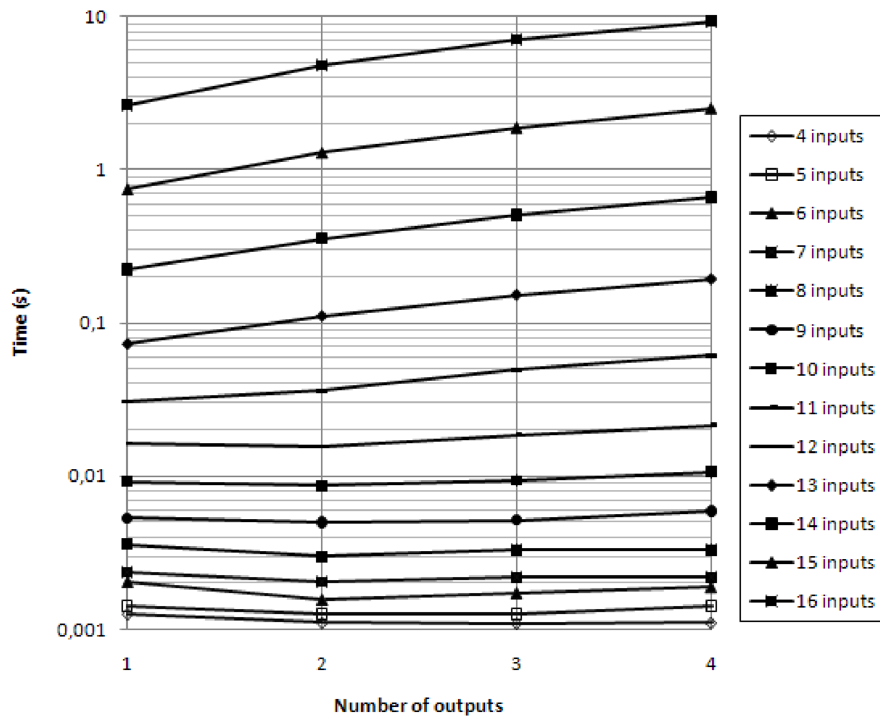

Figure 6.6: Execution time of matching algorithm varying the number of outputs.

(45%) is proportionally larger than the increase in the number of inputs (23%). This indicates that the *KL*-cuts with unbounded K are indeed reconvergent graphs and not single trees.

When performing a greedy covering using all cuts, and a *KL*-cuts with unbounded K and $L = 1$ partitioning, the time consumed dropped, in average, to one third of the time taken without the partitioning, at the expense of 16% more LUTs used.

Table 6.8: Comparison between a covering with factor trees and a covering with *KL*-cuts with unbounded K.

Name	Factor Trees Total	Factor Trees Size	Factor Trees Mean K	*KL*-Cuts with unbounded K Total	*KL*-Cuts with unbounded K Size	*KL*-Cuts with unbounded K Mean K
C1355	164	2.39	2.11	68	5.76	3.68
C1908	157	2.29	2.34	103	3.50	3.02
C2670	134	4.04	3.81	59	9.17	6.39
C3540	230	4.02	4.25	192	4.82	4.86
C5315	350	3.84	3.63	241	5.58	4.58
C6288	1404	1.33	2.33	1403	1.33	2.33
C7552	607	2.34	2.59	370	3.83	3.35
s13207	661	3.12	3.59	646	3.19	3.65
s15850	937	2.94	3.34	848	3.25	3.58
s35932	3355	2.71	2.63	2320	3.91	3.30
s38417	2426	3.35	3.60	2241	3.63	3.76
s38584	2963	3.32	3.58	2531	3.88	4.00
Avg.	1116	2.97	3.15	919	4.32	3.88

7 CONCLUSIONS AND FUTURE WORK

The main contribution of this thesis was the introduction of the concept of *KL*-feasible cuts, which allows controlling both the number K of inputs and the number L of outputs in the computation of circuit cuts. Algorithms for computing *KL*-feasible cuts were presented and results have shown the usefulness of the method. The concept of factor cuts was also extended to *KL*-cuts, which has shown the viability of computing *KL*-cuts with larger K and L. A novel algorithm for computing *KL*-feasible cuts with unbounded K was presented, which is especially useful in partitioning. Similarly, *KL*-cuts with unbounded L were proposed, which can be useful in technology mapping, although not widely explored in this work.

The second contribution was the proposal of two types of covering algorithms. One very simple, that was intended to be a proof of concept algorithm, which is the greedy algorithm, has shown interesting results, even when compared to the state of the art. Its drawback is that it can deal with area only, not being able to treat delay. The second one is an extension of the area flow covering that is able to deal with multiple output cuts. This algorithm is still under investigation, and the results show that there is still place for improvement, as, even with a low utilization of multiple output resources, it has produced fairly good results.

It is worthy to highlight that both algorithms were implemented in a tool, which is able to read an AIG on AIGER format, and write the mapped circuit in a Verilog file, along with a description of a library needed. The use of this standard file formats will make easier to perform tests in a commercial tool environment.

A third contribution was the development of a Boolean matching mechanism that is able to deal with multiple output blocks. It was strongly based on a previous work, but the extension added to the solution make it much more general.

There is still much work that needs to be carried on. First of all, other applications must be explored, such as peephole optimizations — especially AIG rewriting —, regularity extraction and IPO of an already mapped circuit.

The proposed algorithm for covering is still immature. New modes of operation could improve the quality of the results, and further study on partitioning could improve its throughput. There are also some properties that should be simple to implement, but require further analysis. One example is on the exploration of factor cuts in the area flow for multiple outputs algorithm to improve performance. Another point is on implementing the depth minimum and depth constrained mappings, which are supported by the original area flow algorithm.

It was explored in this thesis the application of covering focused in FPGA technology. It is necessary the study on how to switch to standard cell mapping, using this methodology. One of the required infrastructures that were explored in this thesis is the Boolean

matching. To make it more generic, the Boolean matching algorithm should be extended from P-equivalence checking to NPN-equivalence checking.

Even though much work needs to be done, current results have shown the viability and usefulness of *KL*-cuts on the logic synthesis when multiple output blocks are available.

REFERENCES

ABOUZEID, P.; LEVEUGLE, R.; SAUCIER, G. Logic Synthesis for Automatic Layout. In: IFIP WG10.2/WG10.5 WORKSHOPS ON SYNTHESIS FOR CONTROL DOMINATED CIRCUITS, Amsterdam, The Netherlands, The Netherlands. **Proceedings...** North-Holland Publishing Co., 1993. p.335–343.

BERKELAAR, M.; JESS, J. Technology mapping for standard-cell generators. In: COMPUTER-AIDED DESIGN, 1988. ICCAD-88. DIGEST OF TECHNICAL PAPERS., IEEE INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 1988. p.470 –473.

Berkeley Logic Synthesis and Verification Group. **ABC**: a system for sequential synthesis and verification. Available at `http://www.eecs.berkeley.edu/~alanmi/abc`. Accessed in sep. 2010.

BRYANT, R. Graph-Based Algorithms for Boolean Function Manipulation. **Computers, IEEE Transactions on**, [S.l.], v.C-35, n.8, p.677 –691, aug. 1986.

CHATTERJEE, S. et al. Reducing Structural Bias in Technology Mapping. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, [S.l.], v.25, n.12, p.2894 –2903, dec. 2006.

CHATTERJEE, S.; MISHCHENKO, A.; BRAYTON, R. Factor Cuts. In: COMPUTER-AIDED DESIGN, 2006. ICCAD '06. IEEE/ACM INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2006. p.143–150.

CONG, J.; DING, Y. Combinational logic synthesis for LUT based field programmable gate arrays. **ACM Trans. Des. Autom. Electron. Syst.**, New York, NY, USA, v.1, n.2, p.145–204, 1996.

CONG, J.; WU, C.; DING, Y. Cut ranking and pruning: enabling a general and efficient fpga mapping solution. In: FPGA '99: PROCEEDINGS OF THE 1999 ACM/SIGDA SEVENTH INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, New York, NY, USA. **Proceedings...** ACM, 1999. p.29–35.

CORREIA, V.; REIS, A. Advanced technology mapping for standard-cell generators. In: SBCCI '04: PROCEEDINGS OF THE 17TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, New York, NY, USA. **Proceedings...** ACM, 2004. p.254–259.

COSOROABA, A.; RIVOALLON, F. **Achieving Higher System Performance with the Virtex-5 Family of FPGAs**. [S.l.]: Xilinx, 2006.

DARRINGER, J. A. et al. Logic synthesis through local transformations. **IBM J. Res. Dev.**, Riverton, NJ, USA, v.25, n.4, p.272–280, 1981.

DEBNATH, D.; SASAO, T. Efficient computation of canonical form for Boolean matching in large libraries. In: ASP-DAC '04: PROCEEDINGS OF THE 2004 ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, Piscataway, NJ, USA. **Proceedings. . .** IEEE Press, 2004. p.591–596.

DETJENS, E. et al. Technology Mapping in MIS. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings. . .** [S.l.: s.n.], 1987. p.116–119.

GREGORY, D. et al. SOCRATES: a system for automatically synthesizing and optimizing combinational logic. In: DAC: PAPERS ON TWENTY-FIVE YEARS OF ELECTRONIC DESIGN AUTOMATION, 25., New York, NY, USA. **Proceedings. . .** ACM, 1988. p.580–586.

HASSOUN, S.; SASAO, T. (Ed.). **Logic Synthesis and Verification**. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

HINSBERGER, U.; KOLLA, R. Boolean matching for large libraries. In: DAC '98: PROCEEDINGS OF THE 35TH ANNUAL DESIGN AUTOMATION CONFERENCE, New York, NY, USA. **Proceedings. . .** ACM, 1998. p.206–211.

HUTTON, M. et al. **Improving FPGA Performance and Area Using an Adaptive Logic Module**. [S.l.]: Altera, 2004.

JIANG, Y.; SAPATNEKAR, S. S.; BAMJI, C. Technology mapping for high-performance static CMOS and pass transistor logic designs. **IEEE Trans. Very Large Scale Integr. Syst.**, Piscataway, NJ, USA, v.9, n.5, p.577–589, 2001.

KARANDIKAR, S. K.; SAPATNEKAR, S. S. Logical effort based technology mapping. In: ICCAD '04: PROCEEDINGS OF THE 2004 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2004. p.419–422.

KEUTZER, K. DAGON: technology binding and local optimization by dag matching. In: DAC '87: PROCEEDINGS OF THE 24TH ACM/IEEE DESIGN AUTOMATION CONFERENCE, New York, NY, USA. **Proceedings. . .** ACM, 1987. p.341–347.

KHETERPAL, V. et al. Design methodology for IC manufacturability based on regular logic-bricks. In: DAC '05: PROCEEDINGS OF THE 42ND ANNUAL DESIGN AUTOMATION CONFERENCE, New York, NY, USA. **Proceedings. . .** ACM, 2005. p.353–358.

KUKIMOTO, Y.; BRAYTON, R. K.; SAWKAR, P. Delay-optimal technology mapping by DAG covering. In: DAC '98: PROCEEDINGS OF THE 35TH ANNUAL DESIGN AUTOMATION CONFERENCE, New York, NY, USA. **Proceedings. . .** ACM, 1998. p.348–351.

LEHMAN, E. et al. Logic decomposition during technology mapping. In: ICCAD '95: PROCEEDINGS OF THE 1995 IEEE/ACM INTERNATIONAL CONFERENCE ON

COMPUTER-AIDED DESIGN, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1995. p.264–271.

LIEM, C.; LEFEBVRE, M. A constructive matching algorithm for cell generator based technology mapping. In: CIRCUITS AND SYSTEMS, 1992. ISCAS '92. PROCEEDINGS., 1992 IEEE INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.: s.n.], 1992. v.6, p.2965 –2968 vol.6.

LING, A.; ZHU, J.; BROWN, S. Scalable Synthesis and Clustering Techniques Using Decision Diagrams. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, [S.l.], v.27, n.3, p.423–435, march 2008.

MAILHOT, F.; DE MICHELI, G. Algorithms for technology mapping based on binary decision diagrams and on Boolean operations. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, [S.l.], v.12, n.5, p.599 –620, may 1993.

MANOHARARAJAH, V.; BROWN, S.; VRANESIC, Z. Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, [S.l.], v.25, n.11, p.2331–2340, nov. 2006.

MARQUES, F. S. et al. DAG based library-free technology mapping. In: GLSVLSI '07: PROCEEDINGS OF THE 17TH ACM GREAT LAKES SYMPOSIUM ON VLSI, New York, NY, USA. **Proceedings...** ACM, 2007. p.293–298.

MARTINELLO, O. et al. KL-Cuts. In: INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS. **Proceedings...** [S.l.: s.n.], 2009.

MARTINELLO, O. et al. KL-Cuts: a new approach for logic synthesis targeting multiple output blocks. In: DESIGN, AUTOMATION TEST IN EUROPE CONFERENCE EXHIBITION (DATE), 2010. **Proceedings...** [S.l.: s.n.], 2010. p.777–782.

MICHELI, G. D. **Synthesis and Optimization of Digital Circuits**. [S.l.]: McGraw-Hill Higher Education, 1994.

MISHCHENKO, A.; BRAYTON, R. Scalable Logic Synthesis using a Simple Circuit Structure. In: INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS. **Proceedings...** [S.l.: s.n.], 2006.

MISHCHENKO, A.; BRAYTON, R.; CHATTERJEE, S. Boolean factoring and decomposition of logic networks. In: COMPUTER-AIDED DESIGN, 2008. ICCAD 2008. IEEE/ACM INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2008. p.38–44.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. **FRAIGs**: a unifying representation for logic synthesis and verification. [S.l.]: UC Berkeley, 2005.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: DESIGN AUTOMATION CONFERENCE, 2006 43RD ACM/IEEE. **Proceedings...** [S.l.: s.n.], 2006. p.532–535.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. K. Improvements to Technology Mapping for LUT-Based FPGAs. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, [S.l.], v.26, n.2, p.240–253, feb. 2007.

MISHCHENKO, A. et al. Scalable don't-care-based logic optimization and resynthesis. In: FPGA '09: PROCEEDING OF THE ACM/SIGDA INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, New York, NY, USA. **Proceedings...** ACM, 2009. p.151–160.

MVSIS Group. **MVSIS**: multi-valued logic synthesis system. Available at `http://embedded.eecs.berkeley.edu/mvsis/`. Accessed in sep. 2010.

PAN, P.; LIN, C.-C. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In: FPGA '98: PROCEEDINGS OF THE 1998 ACM/SIGDA SIXTH INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, New York, NY, USA. **Proceedings...** ACM, 1998. p.35–42.

REIS, A. I. Covering Strategies for Library Free Technology Mapping. **Integrated Circuit Design and System Design, Symposium on**, Los Alamitos, CA, USA, v.0, p.0180, 1999.

ROSIELLO, A. et al. A Hash-based Approach for Functional Regularity Extraction During Logic Synthesis. In: VLSI, 2007. ISVLSI '07. IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON. **Proceedings...** [S.l.: s.n.], 2007. p.92–97.

SASAO, T. **Switching Theory for Logic Synthesis**. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

SCHNEIDER, F. R. et al. Exact lower bound for the number of switches in series to implement a combinational logic cell. In: ICCD '05: PROCEEDINGS OF THE 2005 INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2005. p.357–362.

SECHEN, C. et al. Libraries: lifejacket or straitjacket. In: DAC '03: PROCEEDINGS OF THE 40TH ANNUAL DESIGN AUTOMATION CONFERENCE, New York, NY, USA. **Proceedings...** ACM, 2003. p.642–643.

SENTOVICH, E. M. et al. **SIS**: a system for sequential circuit synthesis. [S.l.: s.n.], 1992.

STOK, L.; IYER, M. A.; SULLIVAN, A. J. Wavefront technology mapping. In: DATE '99: PROCEEDINGS OF THE CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, New York, NY, USA. **Proceedings...** ACM, 1999. p.108.

VUJKOVIC, M.; SECHEN, C. Optimized power-delay curve generation for standard cell ICs. In: ICCAD '02: PROCEEDINGS OF THE 2002 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, New York, NY, USA. **Proceedings...** ACM, 2002. p.387–394.

WERBER, J.; RAUTENBACH, D.; SZEGEDY, C. Timing optimization by restructuring long combinatorial paths. In: COMPUTER-AIDED DESIGN, 2007. ICCAD 2007. IEEE/ACM INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2007. p.536–543.

A APPENDIX <KL-CUTS: UMA NOVA ABORDAGEM PARA SÍNTESE LÓGICA UTILIZANDO BLOCOS COM MÚLTIPLAS SAÍDAS>

A.1 Introdução

Tecnologias baseadas em circuitos integrados digitais têm grande impacto na sociedade, estando presente em praticamente todas as áreas do conhecimento. Os avanços no campo da concepção de circuitos integrados possibilitam a agregação de um número cada vez maior de componentes em um mesmo dispositivo. Esta elevada escala de integração impõe novos desafios ao processo de síntese. A fim de lidar com constantes mudanças nas regras de projeto, e aumentar a produtividade, a automatização deste processo através da utilização de ferramentas de EDA (do inglês *Electronic Design Automation*) desempenha um papel crucial.

O objetivo da síntese é transformar uma descrição de alto nível de abstração de um circuito em um modelo mais detalhado, como um modelo geométrico. O processo de síntese é frequentemente dividido em três etapas principais: síntese arquitetural, síntese lógica e síntese física (MICHELI, 1994). O papel da síntese lógica é o de traduzir uma descrição lógica de um circuito em uma rede de células de uma determinada tecnologia interligadas. Ela é geralmente dividida em três fases: otimizações independentes de tecnologia, mapeamento tecnológico e otimizações dependentes de tecnologia. Na primeira aplica-se algumas transformações que não dependem da tecnologia, mas dependem do algoritmo de mapeamento escolhido. Estas transformações podem ser estruturais ou Booleanas. Em seguida, a fase de mapeamento tecnológico liga o circuito à tecnologia, substituindo porções do circuito por células implementadas na tecnologia alvo. Depois disso, mais otimizações são aplicadas ao circuito mapeado, como redimensionamento ou duplicação de células lógicas. Estas são chamadas de otimizações dependentes de tecnologia.

Alguns dos recentes avanços na síntese lógica são baseados em AIG (do inglês *And-Inverter Graph*), por motivos de escalabilidade, já que esta é uma estrutura simples e regular (LING; ZHU; BROWN, 2008; MISHCHENKO; BRAYTON, 2006). Parte desses avanços é baseada no conceito de cortes-*K* (CONG; WU; DING, 1999; PAN; LIN, 1998), incluindo algoritmos de re-síntese com base em reescrita de AIG (MISHCHENKO; CHATTERJEE; BRAYTON, 2006). A escalabilidade é obtida mantendo-se o valor de K pequeno de modo que funções lógicas possam ser manipuladas como vetores de inteiros. Por exemplo, em (MISHCHENKO; BRAYTON; CHATTERJEE, 2008) escalabilidade é atingida utilizando funções de 16 ou menos entradas representadas como tabelas verdade.

Algoritmos para computação eficiente de cortes para uma única saída são bem conhecidos. Particularmente, os algoritmos para a computação exaustiva de cortes-*K* foram

introduzidas por Cong (CONG; WU; DING, 1999) e Pan (PAN; LIN, 1998). Chatterjee (CHATTERJEE; MISHCHENKO; BRAYTON, 2006) introduziu o conceito de fatoração de cortes, onde enumeração exaustiva é evitada por fazer uma separação entre nodos árvore e nodos dag no AIG. O cálculo da fatoração de cortes permite trabalhar com cortes de até 16 entradas, o que não é possível com os algoritmos anteriores de enumeração exaustiva. Todos estes algoritmos para enumeração de cortes só são capazes de levar em conta o número K de entradas, não contemplando os benefícios da utilização de múltiplas saídas. Por exemplo, no mapeamento tecnológico usando cortes-K, duplicação da lógica pode ocorrer durante a etapa de cobertura, o que é provavelmente um problema no fluxo de projeto.

Mesmo que as tecnologias atuais suportem blocos com mais de uma saída, como FPGAs (HUTTON et al., 2004; COSOROABA; RIVOALLON, 2006), todo o fluxo está orientado a blocos com uma única saída, e uma etapa de combinação é adicionado no final para tentar tirar vantagem destes elementos com múltiplas saídas.

A.2 Cortes-KL

Cortes são uma maneira eficiente de representar uma região de um AIG quando se trata da geração de um sinal por vez. No entanto, quando se trata de regiões com múltiplas saídas múltiplos cortes seriam necessários. Para superar essa limitação, os propostos *cortes-KL* são subgrafos que não somente têm um número limitado e bem controlado de entradas, mas essas mesmas propriedades são estendidas para as saídas.

Em essência, um corte-*KL* define uma região de um grafo, que tenha no máximo K entradas e, no máximo, L saídas. É representado pelo conjunto de nodos entrada e um conjunto de nodos saída. Todos os nós entre o conjunto de entradas e o conjunto de saídas, incluindo o conjunto de saídas, mas excluindo o conjunto de entradas, estão “dentro” da região delimitada.

A.2.1 Aplicações

A.2.1.1 Mapeamento Tecnológico

Células com mais de uma saída são uma realidade em bibliotecas atuais, como por exemplo, as células somador completo e meio somador. Da mesma forma, mapeamentos livres de tecnologia podem utilizar células com múltiplas saídas para reduzir a área de um circuito, especialmente em circuitos aritméticos. Além disso, FPGAs atuais têm LUTs de múltiplas saídas disponíveis (HUTTON et al., 2004; COSOROABA; RIVOALLON, 2006). Logo a utilização de metodologias que só consideram porções com uma única saída podem levar a um resultado de má qualidade.

A Tabela A.1 mostra alguns resultados de mapeamentos para blocos de uma saída, que servirá como base de comparação para os algoritmos de múltiplas saídas. Dois métodos foram utilizados. O primeiro, mostrado na coluna “ABC”mostra o número de LUTs usadas pelo mapeamento do circuito produzido rodando os comandos ‘*r; dch, if -C 12; mfs -W 4 -M 5000*’ no ABC (MISHCHENKO et al., 2009) quatro vezes, e escolhendo o melhor resultado, com uma biblioteca contendo LUTs de até 5 entradas. Esta biblioteca assume o mesmo custo para cada LUT. Constantes, *buffers* e inversores não são considerados neste valor. O segundo método é o método do fluxo de área (MANOHARARAJAH; BROWN; VRANESIC, 2006), depois de oito iterações e usando $\alpha = 2$. Pode-se notar que o fluxo de área é quase três vezes mais rápido que o ABC, mas produz resultados um

pouco piores.

Table A.1: Cobertura para LUTs de uma saída usando ABC e fluxo de área.

	ABC		Fluxo de Área	
Nome	LUTs	Tempo (s)	LUTs	Tempo (s)
C1355	68	4.11	66	1.27
C1908	105	4.12	101	0.92
C2670	149	2.91	127	1.08
C3540	276	15.30	285	2.13
C5315	324	10.69	339	2.73
C6288	501	51.16	711	7.11
C7552	372	14.05	373	4.48
s13207	718	12.69	707	2.89
s15850	966	16.83	945	4.86
s35932	2493	14.95	2682	16.34
s38417	2659	49.22	2648	17.89
s38584	2655	29.27	2754	19.08
Média	941	18.78	978	6.73

A.2.1.1.1 Mapeamento Guloso

Um algoritmo simples para realizar uma cobertura completa de um circuito com cortes-*KL* foi elaborado. Este algoritmo não pretende atingir o estado-da-arte em mapeamento, mas apenas confirmar o potencial da utilização de cortes-*KL* no mapeamento tecnológico. Este algoritmo guloso procura por máximos locais. Em cada iteração o maior corte-*KL* possível é escolhido e todos os cortes-*KL* com nodos sobrepostos são eliminados do espaço de solução. Estas iterações são repetidas até que o circuito esteja completamente coberto.

A Tabela A.2 mostra os resultados de cobertura do mapeamento guloso. A coluna nomeada "Cortes" mostra o número de cortes da cobertura resultante. Coluna "% CMS" são a percentagem dos cortes que têm múltiplas saídas. Dois cortes com uma saída podem ser implementados por uma LUT de duas saídas, desde que a soma do número de entradas desses cortes não seja maior do que o número de entradas da LUT. Por exemplo, um corte-2-1 sempre pode ser combinado com um corte-3-1, se houver uma LUT-5-2 disponível. Por esse motivo, a coluna "LUTs", que mostra o número de LUTs necessárias para implementar a cobertura, apresenta um valor menor do que a coluna "Cortes". A coluna "% LMS" representa a porcentagem de LUTs usadas que usam as duas saídas.

O algoritmo encontra uma boa fração de cortes *KL* com múltiplas saídas, e a maioria dos de única saída têm poucas entradas, o que permite sua combinação levando a uma elevada utilização de LUTs de múltiplas saídas (mais de 85 %, em média), resultando em redução de LUTs utilizadas (redução de 30 % em comparação com o mapeamento ABC).

Embora essa cobertura gulosa produza resultados com menos LUTs que o ABC, isso não significa que ele é um algoritmo melhor. Deve ser levado em consideração que as LUTs usadas pelo ABC têm uma saída, enquanto a nossa tem duas. Portanto, no caso ideal, o nosso mapeamento deveria produzir uma redução de 50 % no número de LUTs, se o mapeamento tiver qualidade equivalente e se a topologia do circuito permitir essa

Table A.2: Cobertura gulosa usando cortes-*KL*.

Nome	Tempo (s)	Cortes	% CMS	LUTs	% LMS
C1355	0.95	74	37.84	54	88.89
C1908	0.72	75	69.33	64	98.44
C2670	1.16	141	35.46	110	73.64
C3540	2.38	273	47.25	223	80.27
C5315	3.42	326	44.48	269	75.09
C6288	5.06	359	91.36	344	99.71
C7552	3.70	333	66.07	285	94.04
s13207	5.20	589	55.69	487	88.30
s15850	8.33	819	47.86	657	84.32
s35932	87.69	2221	55.74	1746	98.11
s38417	70.84	2159	61.32	1809	92.54
s38584	112.66	2631	40.94	2123	74.66
Média	25.18	833	54.45	681	87.33

redução. Por outro lado, se mesmo com esse algoritmo simples os resultados foram relativamente bons, isso significa que essa estratégia de considerar blocos de várias saídas desde o início está na direção certa.

A.2.1.1.2 Mapeamento de Fluxo de Área para Múltiplas Saídas

Uma extensão ao algoritmo de cobertura de fluxo de área (MANOHARARAJAH; BROWN; VRANESIC, 2006) é apresentado nesta dissertação, uma vez que o original lida apenas com cortes de única saída. O cálculo do fluxo área permanece praticamente o mesmo. As principais diferenças são a classificação de nodos, e a criação de diferentes modos de operação.

Este algoritmo modificado produziu os resultados mostrados na tabela A.3. A primeira coisa a notar é que a utilização de cortes com duas saídas é muito menor do que os produzidos pela cobertura gulosa. Além disso, o algoritmo tende a selecionar os cortes com um grande número de entradas, tornando difícil a combinação de cortes em uma única LUT de duas saídas. Por outro lado, mesmo com esta sub-utilização das LUTs de múltiplas saídas, o número total de LUTs é menos que 10 % maior do que a cobertura gulosa. Em outras palavras, há mais espaço para melhorias neste algoritmo que no algoritmo de cobertura gulosa.

Um circuito em especial utilizou quase 100 % dos cortes de múltiplas saídas, onde o número de LUTs usadas foi a metade do utilizado pelo ABC. O circuito *c6288* é um circuito muito regular, que implementa um multiplicador 16 por 16 bits como uma matriz de somadores.

Outra vantagem deste algoritmo de fluxo de área sobre a cobertura gulosa é explicado a seguir. O algoritmo original de fluxo de área é capaz de encontrar a cobertura de profundidade mínima, e depois executar a recuperação da área nas próximas iterações. Esta versão múltiplas saídas poderia ter essa característica também, mas a cobertura gulosa não pode.

Table A.3: Mapeamento de fluxo de área para múltiplas saídas.

Nome	Tempo (s)	Cortes	% CMS	LUTs	% LMS
C1355	1.81	71	9.86	64	21.88
C1908	1.44	83	24.10	66	56.06
C2670	1.59	119	9.24	114	14.04
C3540	3.08	252	11.11	235	19.15
C5315	4.20	314	7.32	302	11.59
C6288	15.39	241	98.76	240	99.58
C7552	6.77	302	19.87	298	21.48
s13207	4.09	645	9.61	564	25.35
s15850	6.42	850	10.94	736	28.13
s35932	28.88	2266	23.74	1664	68.51
s38417	25.61	2401	7.71	2168	19.28
s38584	25.84	2501	9.96	2364	16.33
Média	10.43	837	20.18	735	33.45

A.2.1.2 Identificação de Padrões

O processo de cobertura necessita, além da estrutura a ser coberta, um espaço de busca formado por células. Este espaço de busca, ou espaço de solução, é composto por correspondências entre as porções do grafo a ser coberto e uma biblioteca de células predefinida.

Quando o mapeamento usa uma abordagem baseada em bibliotecas, a fase de identificação de padrões consiste em decidir se a biblioteca possui uma célula correspondente ao sub-circuito a ser coberto. Neste contexto, "correspondente" pode ter várias definições. Em uma identificação de padrões estrutural, um sub-circuito corresponde a uma célula se estes forem representados por grafos isomórficos. Em uma identificação de padrões Booleana, a correspondência se dá quando a função booleana implementada pelo sub-circuito pertence à mesma classe de equivalência que a função implementada pela célula.

Bem como os conceitos de classes de equivalência P, NP e NPN são definidas para uma função, as idéias de classes de equivalência PP, NPP ou NPNP podem ser definidas para uma lista de funções, o que pode ser visto como uma função de múltiplas saídas $f : B^n \mapsto B^m$.

Esta dissertação propõe uma extensão do sistema TEMPLATE (HINSBERGER; KOLLA, 1998) a fim de ser capaz de encontrar equivalência-PP entre listas de funções. O foco é a aplicação para determinar equivalência entre células com múltiplas saídas. A ideia é semelhante ao algoritmo original: encontrar uma lista representativa das funções $R[L_f]_{PP}$ de $[L_f]_{PP}$.

O gráfico da Figura A.1 mostra o desempenho do algoritmo desenvolvido. O número de entradas variou de 4 a 16, e o número de saídas de 1 a 4. Cada ponto no gráfico é um valor médio obtido após 100 execuções de funções geradas aleatoriamente. É notório que a dependência do tempo consumido com número de entradas é superior a um exponencial até cerca de $K = 11$, e então torna-se uma linha no gráfico semi-log, o que denota uma exponencial. A linha que representa um corte de única saída é aproximadamente o tempo do algoritmo original. A Figura A.2 mostra os mesmos resultados, mas com L em vez de K no eixo x. Neste gráfico pode-se observar que o tempo de execução também tem uma

relação exponencial com o número de saídas. No entanto, a inclinação é muito menos acentuada, o que significa que é mais barato adicionar uma saída do que adicionar uma entrada.

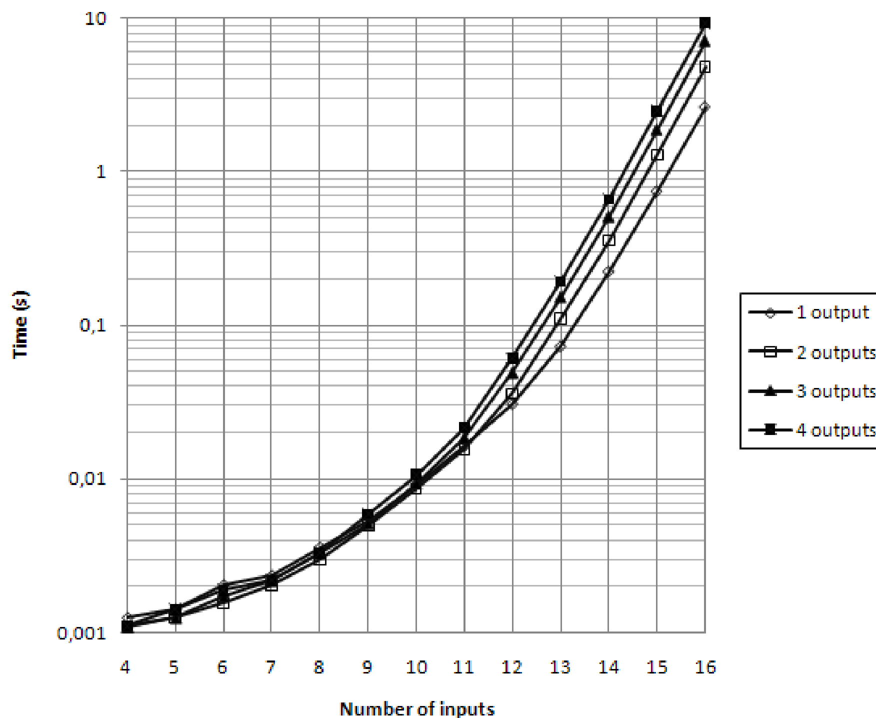

Figure A.1: Tempo de execução de algoritmo de identificação de padrões variando o número de entradas.

A.2.1.3 Particionamento

Algoritmos de mapeamento tecnológico frequentemente fazem uso de heurísticas a fim de reduzir a complexidade do problema. Uma das heurísticas mais populares é o particionamento.

O particionamento consiste em dividir o circuito a ser coberto em várias partes menores, realizando o mapeamento individual em cada parte. A decomposição de um gráfico em uma floresta de árvores é um exemplo de particionamento.

Uma decomposição um pouco mais abrangente do que a quebra em árvores é o particionamento do circuito em MFFCs. Pode-se utilizar algoritmos de enumeração de cortes-*KL* para encontrar MFFCs ou algo mais geral.

A.3 Conclusões

A principal contribuição desta dissertação é a introdução do conceito de cortes-*KL*, o que permite controlar tanto o número K de entradas e o número L de saídas no cálculo dos cortes do circuito. Algoritmos para computação cortes-*KL* são apresentados e os resultados demonstram a utilidade do método.

A segunda contribuição foi a proposta de dois tipos de algoritmos de cobertura. Um muito simples, que serve como prova de conceito, que é o algoritmo guloso, mostrou resultados interessantes, mesmo quando comparado com o estado da arte. Sua desvantagem é que ele pode lidar unicamente com área, não sendo capaz de tratar atraso. A segunda é

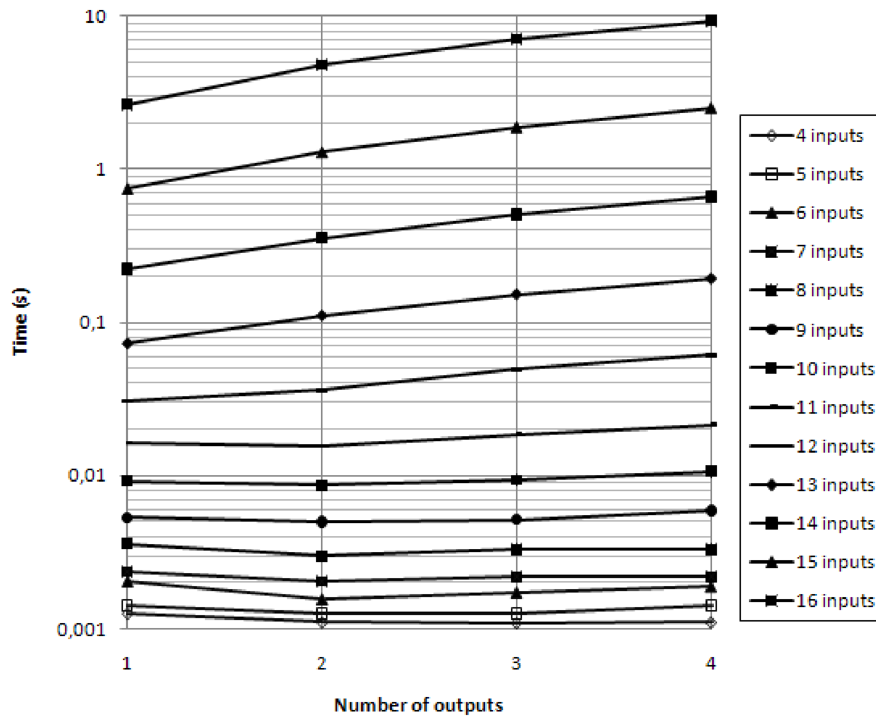

Figure A.2: Tempo de execução de algoritmo de identificação de padrões variando o número de saídas.

uma extensão do algoritmo de cobertura de fluxo de área que é capaz de lidar com cortes com múltiplas saídas. Este algoritmo ainda está sob investigação, e os resultados mostram que ainda há lugar para melhorias, mas mesmo com uma baixa utilização de recursos de múltiplas saídas, tem produzido resultados bastante bons.

Uma terceira contribuição foi o desenvolvimento de um mecanismo Booleano de identificação de patrões que é capaz de lidar com blocos de múltiplas saída. Foi fortemente baseada em um trabalho anterior, mas a extensão adicionada à solução a torna muito mais geral.

Ainda há muito trabalho a ser desenvolvido. Primeiro de tudo, outras aplicações devem ser explorados, como otimizações locais — especialmente reescrita de AIG —, extração de regularidade e IPO (do inglês *In-Place Optimization*) de um circuito já mapeado.

O algoritmo proposto para a cobertura ainda é imaturo. Novos modos de operação poderiam melhorar a qualidade dos resultados, e um estudo mais aprofundado sobre particionamento poderia melhorar o seu rendimento. Existem também algumas propriedades que devem ser simples de implementar, mas exigem uma análise mais aprofundada. Um exemplo é sobre mapeamento com profundidade lógica mínima e com profundidade limitada, que são suportados pelo algoritmo original de fluxo de área.

Mesmo tendo muito trabalho a ser feito, os resultados atuais têm demonstrado a viabilidade e utilidade de cortes-*KL* na síntese lógica, quando blocos de múltiplas saídas estão disponíveis.