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

LUCAS MACHADO

KL-Cut Based Remapping

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Microelectronics

Prof. Dr. André Inácio Reis
Advisor

Prof. Dr. Renato Perez Ribas
Co-advisor

Porto Alegre, May 2013.

CIP – CATALOGING-IN-PUBLICATION

Machado, Lucas

KL-Cut Based Remapping / Lucas Machado. – Porto Alegre: PGMICRO da UFRGS, 2013.

88 f.:il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR – RS, 2013. Advisor: André Inácio Reis; Co-advisor: Renato Perez Ribas.

1. Digital Circuits. 2. Logic Synthesis. 3. Technology Remapping. 4. Cut Enumeration. 5. KL-cuts. 6. Multiple output blocks. I. Reis, André Inácio. II. Ribas, Renato Perez. III. KL-Cut Based Remapping.

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 PGMICRO: Prof. Ricardo Augusto da Luz Reis
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Engineers like to solve problems.
If there are no problems handily available,
they will create their own problems to solve.”
– Dilbert

ACKNOWLEDGEMENTS

I thank my advisor André Reis for his ideas, support, good critics and sense of humor, which guided me during the development of this work. Also, I need to thank my co-advisor Renato Ribas, for his understanding, encouraging, personal and professional guidance, which helped me since I was an undergraduate student.

I am grateful for the help and support of the whole LogiCS group, but especially for some people: Osvaldo Martinello, who performed the work that inspired mine; Mayler Martins, which work on logic minimization (and all the possible variations we discussed) was crucial to get the results I had; Vinicius Callegaro, who helped me a lot on Java development, and implemented the mapping tool (with the very specific details) used in this work; and Oendel Merlo, who implemented the SDC parser in all variations I asked, which is also used in this work.

Also, I thank my friends from Lajeado and from Porto Alegre, which were there for me in all times, during school, during graduation, during my research time, and helped me to be the person I am today.

I need to thank immensely my family: my father Carson Machado, my mother Gisele Machado and my brother Jonas Machado. I missed them a lot during the time I was out, but they were, are and will always be my support to everything, and the giants that took me in their shoulders and made me look further.

Last but not least, my girlfriend Rafaela Bortolini has my deepest gratitude. She gave me the encouragement, the understanding, the love and caring more than I needed. She gave me the north and help that I needed in life, and also to finish this work.

This research was partially funded by Nangate Inc. under a Nangate/UFRGS research agreement, by CAPES and CNPq funding agencies, by FAPERGS under grant 11/2053-9 (Pronem), and by the European Community's Seventh Framework Programme under grant 248538 – Synaptic.

ABSTRACT

This work introduces the concept of *k*-cuts and *kl*-cuts on top of a mapped circuit in a netlist representation. Such new approach is derived from the concept of *k*-cuts and *kl*-cuts on top of AIGs (and inverter graphs), respecting the differences between these two circuit representations. The main differences are: (1) the number of allowed inputs for a logic node, and (2) the presence of explicit inverters and buffers in the netlist. Algorithms for enumerating *k*-cuts and *kl*-cuts on top of a mapped circuit are proposed and implemented. The main motivation to use *kl*-cuts on top mapped circuits is to perform local optimization in digital circuit logic synthesis.

The main contribution of this work is a novel iterative remapping approach using *kl*-cuts, reducing area while keeping the timing constraints attained. The use of complex gates can potentially reduce the circuit area, but they have to be chosen wisely to preserve timing constraints. Logic synthesis commercial design tools work better with simple cells and are not capable of taking full advantage of complex cells. The proposed iterative remapping approach can exploit a larger amount of logic gates, reducing circuit area, and respecting global timing constraints by performing an STA (static timing analysis) check. Experimental results show that this approach is able to reduce up to 38% in area of the combinational portion of circuits for a subset of IWLS 2005 benchmarks, when compared to results obtained from logic synthesis commercial tools.

Another contribution of this work is a novel yield model for digital integrated circuits (IC) manufacturing, considering lithography printability problems as a source of yield loss. The use of regular layouts can improve the lithography, but it results in a significant area overhead by introducing regularity. This is the first approach that considers the tradeoff of cells with different level of regularity and different area overhead during the logic synthesis, in order to improve overall design yield. The technology remapping tool based on *kl*-cuts developed was modified in order to use such yield model as cost function, improving the number of good dies per wafer, with promising interesting results.

Keywords: Digital circuits, logic synthesis, technology mapping, cut enumeration, static timing analysis, remapping, lithography.

Remapeamento baseado em cortes KL

RESUMO

Este trabalho introduz o conceito de cortes *k* e cortes *kl* sobre um circuito mapeado, em uma representação *netlist*. Esta nova abordagem é derivada do conceito de cortes *k* e cortes *kl* sobre AIGs (*and inverter graphs*), respeitando as diferenças entre essas duas formas de representar um circuito. As principais diferenças são: (1) o número de entradas em um nodo do grafo, e (2) a presença de inversores e *buffers* de forma explícita no circuito mapeado. Um algoritmo para enumerar cortes *k* e cortes *kl* é proposto e implementado. A principal motivação de usar cortes *kl* sobre circuitos mapeados é para realizar otimizações locais na síntese lógica de circuitos digitais.

A principal contribuição deste trabalho é uma abordagem nova de remapeamento iterativo, utilizando cortes *kl*, reduzindo a área do circuito e respeitando as restrições de temporização do circuito. O uso de portas lógicas complexas pode potencialmente reduzir a área total de um circuito, mas elas precisam ser escolhidas corretamente de forma a manter as restrições de temporização do circuito. Ferramentas comerciais de síntese lógica trabalham melhor com portas lógicas simples e não são capazes de explorar eventuais vantagens em utilizar portas lógicas complexas. A abordagem proposta de remapeamento iterativo utilizando cortes *kl* é capaz de explorar uma quantidade maior de portas lógicas com funções lógicas diferentes, reduzindo a área do circuito, e mantendo as restrições de temporização intactas ao fazer uma checagem STA (análise temporal estática). Resultados experimentais mostram uma redução de até 38% de área na parte combinacional de circuitos para um subconjunto de *benchmarks* IWLS 2005, quando comparados aos resultados de ferramentas comerciais de síntese lógica.

Outra contribuição deste trabalho é um novo modelo de rendimento (*yield*) para fabricação de circuitos integrados (IC) digitais, considerando problemas de resolução da etapa de litografia como uma fonte de diminuição do *yield*. O uso de leiautes regulares pode melhorar bastante a resolução da etapa de litografia, mas existe um aumento de área significativo ao se introduzir a regularidade. Esta é a primeira abordagem que considera o compromisso (*trade off*) de portas lógicas com diferentes níveis de regularidade e diferentes áreas durante a síntese lógica, de forma a melhorar o *yield* do projeto. A ferramenta desenvolvida de remapeamento tecnológico utilizando cortes *kl* foi modificada de forma a utilizar esse modelo de yield como função custo, de forma a aumentar o número de boas amostras (*dies*) por lâmina de silício (*wafer*), com resultados promissores.

Palavras-Chave: circuitos digitais, síntese lógica, mapeamento tecnológico, enumeração de cortes, análise temporal estática, remapeamento, litografia.

LIST OF FIGURES

Figure 1.1 – Picture of the Moore's "Law" in the Computer History Museum showing the number of dies per wafer in linear scale, California, United States of America (June, 2012) ... 19
Figure 1.2: Logic synthesis in the standard cell design flow ... 21
Figure 2.1: Truth tables representing the (a) AND, (b) OR and (c) NOT logical operations. ... 26
Figure 2.2: Truth table representing function f ... 26
Figure 2.3: Schematic representation of (a) NOT, (b) AND and (c) OR operators. ... 28
Figure 2.4: Boolean network mapped from Equation (2.3) ... 28
Figure 2.5: Boolean network mapped from Equation (2.5) ... 28
Figure 2.6: Boolean functions in the same P-equivalence class. ... 29
Figure 2.7: Boolean functions in the same NP-equivalence class. ... 29
Figure 2.8: Boolean functions in the same NPN-equivalence class. ... 29
Figure 2.9: Venn diagram showing the relationship between Boolean functions equivalence classes: P, NP and NPN. ... 30
Figure 2.10: An example of directed acyclic graph ... 30
Figure 2.11: Example of AND-inverter graph. ... 31
Figure 2.12: Example of mapped circuit. ... 31
Figure 2.13: An example of a logic tree. ... 32
Figure 2.14 An AIG example illustrating covering using kl-cuts. Nodes a, b, c, d, 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. (MARTINELLO, 2010) ... 33
Figure 2.15: Process to obtain cell delay and output transition time through the NLDM, extracting the values from a Liberty file. It is necessary to know the input transition time and the output load, and perform a bilinear interpolation in the values read in the Liberty file. ... 36
Figure 3.1 Exaustive approach for computing $R[f]_P$ (HINSBERGER; KOLLA, 1988) 39
Figure 3.2 Reducing search space by cutting non-maximal branches (HINSBERGER; KOLLA, 1998) ... 40
Figure 3.3 Example of Boolean factoring using Functional Composition. (MARTINS, 2012) ... 41
Figure 3.4 Cost calculation and the first cut of the tree removed. (CORREIA, 2004) .. 42
Figure 3.5 Example of logic tree covering (before inverter minimization). (CORREIA, 2004) ... 42
Figure 4.1 Combinational circuit example to demonstrate k-cuts and kl-cuts computation. ... 44
Figure 4.2 And-inverter graph (AIG) representing a circuit. ... 45
Figure 4.3 Mapped circuit structurally similar to the AIG of Figure 4.2. ... 46
Figure 4.4 Pseudo-code for kl-cuts enumeration on top of a mapped circuit ... 46

Figure 4.5 Example of kl-cut found in a commercial benchmark. 47
Figure 4.6 Example of Figure 4.3 remapped with polarity don't cares information...... 47
Figure 4.7 Example of Figure 4.3 remapped without polarity don't cares information. 48
Figure 4.8 Logic circuit example.. 48
Figure 5.1 Proposed kl-cut remapping flow. 49
Figure 5.2 Example of a complete remapping script....................................... 50
Figure 5.3 Cell information read from a Liberty file...................................... 51
Figure 5.4 Example of a library with four different logic gates. 52
Figure 5.5 Example of structural Verilog. 52
Figure 5.6 Example of the circuit data structure. 53
Figure 5.7 Example of kl-cut data structure representation............................. 54
Figure 5.8 Example of kl-cut input in the remapping approach. 56
Figure 5.9 Example of Figure 5.8 remapped by the proposed approach. 57
Figure 5.10 Report timing performed in the c432 benchmark. 59
Figure 6.1 Comparison between (a) designed layout and (b) lithography simulation of the designed layout. 62
Figure 6.2 Passing technology information to the remapping tool.................. 68
Figure 6.3 (a) Standard "yield-aware" flow, (b) Flow proposed by Nardi (2004), and (c) the flow propose by this work. 68

LIST OF TABLES

Table 2.1: The k-cuts for all nodes of the AIG shown in Figure 2.11........................... 33
Table 3.1 Relationships allowed in multi-level logic factoring..................................... 40
Table 4.1 All k-cuts with k=5 for all nodes of the combinational circuit example of Figure 4.1.. 44
Table 5.1 Commands currently read in the STA engine developed............................. 54
Table 5.2 Original and factorized logic expressions of the kl-cut remapping example. 56
Table 5.3 Different paths analysed in the STA check.. 58
Table 6.1 Values of CHSci derived according to Equation (6.13) considering a wafer of 600 cm².. 66
Table 6.2. Values of #GDW considering a die of 4 cm² on a wafer of 600 cm² for the reference cells, and the same number of cell instances for 1D-restr and 2D-restr cells; sld=1. 67
Table 7.1 Comparison of the worst delay given by a commercial STA tool and the STA engine developed for this work. 72
Table 7.2 Libraries used for area reduction experiments. 72
Table 7.3 ISCAS'85 benchmarks synthesized with commercial tool A. 73
Table 7.4 ISCAS'85 benchmarks synthesized with commercial tool B......................... 74
Table 7.5 ISCAS'85 benchmarks synthesized with commercial tool A remapped with the proposed methodology. 74
Table 7.6 ISCAS'89 benchmarks synthesized with commercial tool A. 75
Table 7.7 ISCAS'89 benchmarks synthesized with commercial tool A remapped with the proposed methodology. 76
Table 7.8 ITC'99 benchmarks synthesized with commercial tool A. 77
Table 7.9 ITC'99 benchmarks synthesized with commercial tool A remapped with with the proposed methodology. 78
Table 7.10 Libraries used for manufacturing improvement experiments. 79
Table 7.11 Manufacturability results for reference library. 80
Table 7.12 Manufacturability results for 2D-gridded library....................................... 80
Table 7.13 Manufacturability results for 1D-restricted library. 81
Table 7.14 Comparison between 1D-restricted library with reference library.............. 81

CONTENTS

ABSTRACT **7**
RESUMO **9**
LIST OF FIGURES **11**
LIST OF TABLES **13**
LIST OF ABBREVIATIONS AND ACRONYMS **17**
1. INTRODUCTION **19**
1.1 Logic synthesis **20**
1.2 Motivation **21**
1.3 Objectives **22**
1.4 Thesis organization **23**
2. TECHNICAL BACKGROUND **25**
2.1 Boolean functions **25**
2.2 Boolean expressions **26**
2.3 Boolean networks **27**
2.4 Equivalence of logic functions **28**
2.5 Data structures **30**
2.5.1 Directed acyclic graphs 30
2.5.2 And-inverter graph 30
2.5.3 Mapped circuit 31
2.5.4 Logic tree 31
2.6 Cuts on AIGs **32**
2.6.1 K-cuts 32
2.6.2 KL-cuts 33
2.7 Cell library **34**
2.8 Technology mapping **34**
2.9 Static Timing Analysis **35**
3. STATE-OF-THE-ART **37**
3.1 Technology mapping **37**
3.2 Boolean matching **38**
3.3 Boolean factoring **39**
3.4 Logic tree mapping **41**
4. CUTS ON MAPPED CIRCUITS **43**
4.1 Differences between AIGs and mapped circuits **43**
4.1.1 Number of nodes inputs 43
4.1.2 Inverters as nodes or edges 43
4.2 K-cuts **44**
4.3 KL-cuts **45**
4.4 Enumeration algorithm **46**
4.5 Polarity *don't cares* **47**

4.6 Degrees of freedom 48
4.7 Conclusion 48
5. KL-CUT BASED REMAPPING 49
5.1 Remapping using KL-cuts 49
5.2 KL-cut remapping flow 50
5.2.1 Liberty parser and data structure 51
5.2.2 Verilog parser and data structure 52
5.2.3 SDC parser and data structure 53
5.2.4 KL-cut enumeration and data structure 53
5.2.5 KL-cut P-group 54
5.2.6 KL-cut remapping approach 55
5.2.7 KL-cut replacement 56
5.2.7.1 KL-cuts overlapping 57
5.2.7.2 STA checking engine 57
5.2.7.3 Greedy choice and iterative remap 58
5.3 Conclusion 58
6. MANUFACTURABILITY COST FUNCTION 61
6.1 Design for manufacturability 61
6.2 Yield background 62
6.3 Yield model 63
6.4 Yield as a cost function 65
6.5 Yield remapping tool 67
6.6 Conclusion 69
7. RESULTS 71
7.1 Precision of STA engine 71
7.2 Area as a cost function 71
7.2.1 Libraries used for area experiments 72
7.2.2 ISCAS benchmarks area results 72
7.2.2.1 ISCAS'85 benchmarks area results 73
7.2.2.2 ISCAS'89 benchmarks area results 74
7.2.3 ITC'99 benchmarks area results 76
7.3 Manufacturability as a cost function 78
7.3.1 Libraries used for manufacturability experiments 79
7.3.2 ISCAS'85 benchmarks manufacturability results 79
8. CONCLUSIONS AND FUTURE WORK 83
8.1 Future work 84
REFERENCES 85

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-inverter graph
ASIC	Application Specific Integrated Circuit
CAD	Computer-Aided Design
CHS	Criticality of Hot Spots
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chemical Metal Polishing
DAG	Directed acyclic graph
DFM	Design For Manufacturability
DH	Design House
EDA	Electronic Design Automation
FC	Functional Composition
GDW	Good Dies per Wafer
IWLS	International Workshop of Logic & Synthesis
LEF	Library Exchange Format
HDL	Hardware Description Language
NLDM	Nonlinear Delay Model
NPN	Negation Permutation Negation
OPC	Optical Proximity Correction
PI	Primary Input
PO	Primary Output
POS	Product of Sums
PSM	Phase Shift Mask
QoR	Quality of Results
RET	Resolution Enhancement Technique
ROBDD	Reduced Ordered Binary Decision Diagram
RTL	Register Transfer Level
SDC	Synopsys Design Constraints
SLD	Severity of Lithography Defects
SOP	Sum of Products
STA	Static Timing Analysis
VLSI	Very-Large-Scale Integration

1. INTRODUCTION

The world has changed a lot in the past fifty years. Most areas of human knowledge, if not all, have improved significantly and these remarkable advancements happened much faster than ever before. Problems that no one could ever think back then are now solved in smartphones. The medicine has great diagnostic machines, a car is able to drive itself taking pictures of the streets, and talking with someone anywhere in the world is not a problem at all. All these achievements in this brave new world have a major source: the integrated circuits (IC). In 1965, Gordon Moore stated that the number of transistors in a chip would double every 24 months (MOORE, 1965), as seen in the Figure 1.1. This trend predicted by Moore guided the evolution of computers and its use in every field. Also, the use of computers to create new and better computers emerged the electronic design automation (EDA) industry, creating a virtuous circle and enabling this fast growing in technology in the recent years.

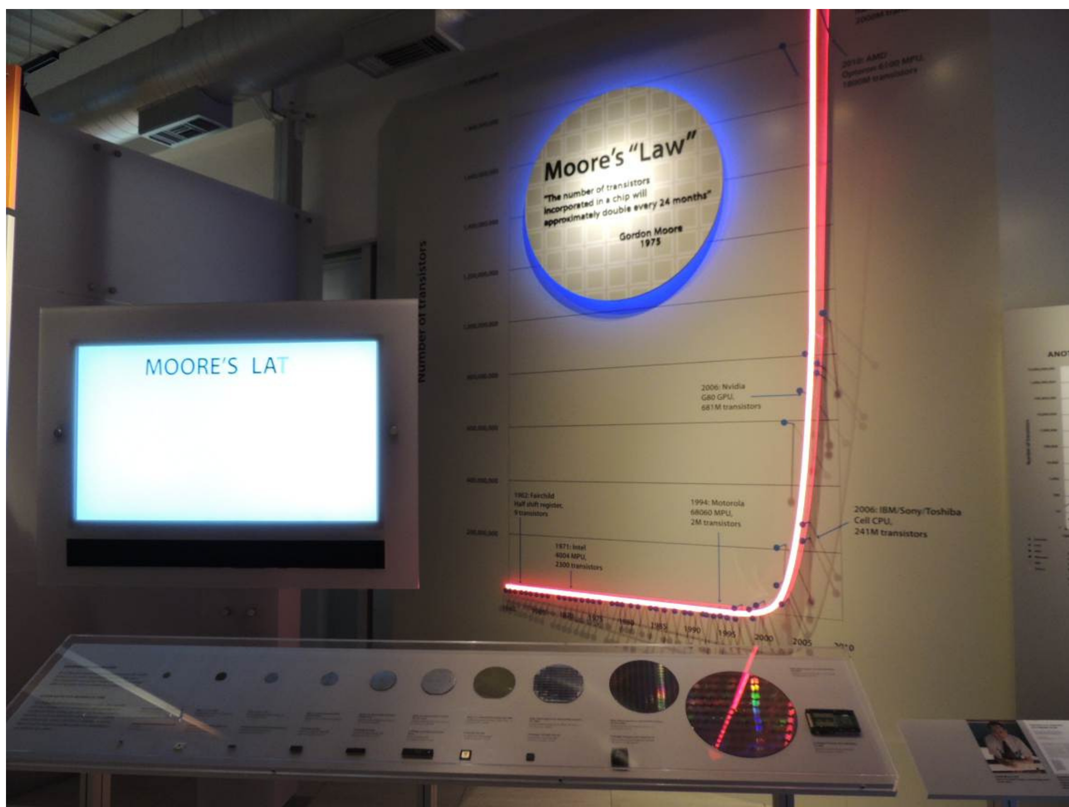


Figure 1.1 – Picture of the Moore's "Law" in the Computer History Museum showing the number of dies per wafer in linear scale, California, United States of America (June, 2012)

Looking at the Figure 1.1, a question hangs in the air: "how long will this trend continue?" The scaling down of the current technology, based on MOS transistors, is reaching a physical limit. The semiconductor most used to manufacture ICs is the Si, and the lattice spacing of a Si crystal is around 0.5 nm (SZE, 2006). This means that there are approximately 10 atoms of Si (or doping elements) within 5 nm. A variability of 1 atom (which is very low) in the manufacturing process of 5 nm transistors will likely produce 10% of variability between transistors (which is quite a lot). This phenomenon is known as discrete random doping and it is just one of several different effects that exist in deep nanometer scaling of current CMOS technology (SZE, 2006). Nowadays, the manufacturing process is about 20 nm and has already lots of obstacles to make it work. Clearly, this trend will likely have an end soon.

Besides CMOS scaling, several research topics investigate the next technology to substitute CMOS, such as quantum computing, graphene, and carbon nanotubes. However, until now, no feasible and effective substitute for CMOS has been found. On the other hand, besides decreasing the size of transistors and investigating new technologies, there are good alternatives that can be investigated and developed in order to produce better and cheaper ICs, even with the current CMOS technology. For instance, developing more powerful EDA tools, with higher quality-of-results (QoR), will improve current IC performance and lower down its costs.

Before EDA tools, ICs were designed by hand. The first microprocessors were drawn in engineering paper and color pencils, and then manufactured in primitive semiconductor planar technology. After the first computers, it was possible to create tools to help with the drawings, and then to place the transistors and route its wires. In mid-80's, the hardware description languages (HDL) emerged, changing completely the way that ICs were designed. The logic synthesis tools starting from RTL descriptions were introduced, trying to obtain the best hardware implementation for a given RTL hardware description. Notice that logic synthesis tasks are very complex since many variables must be taken into account, and trying all possibilities is not computationally feasible. Consequently, in order to obtain good results, within reasonable time-to-market, several heuristics were created, generating sub-optimal results. Therefore, logic synthesis tools still have room for improvements and this work tries to explore this.

1.1 Logic synthesis

Logic synthesis is an important area of study in the field of very large scale integration (VLSI) design automation, being responsible for the transformation of a circuit behavior description into a netlist of logic gates for a given technology, i.e. a digital mapped circuit. The logic synthesis is also an important process in the application specific integrated circuit (ASIC) standard cell design flow, followed by the physical synthesis where the placement and the routing of the logic gates are performed, as illustrated in Figure 1.2.

According to Figure 1.2, the logic synthesis process can be divided into five stages. In the first step, a hardware description is compiled and transformed in a technology independent circuit representation. This circuit representation can be an and-inverter graph (AIG), for instance. Then, several optimizations are performed in this circuit representation. In AIGs, it is important to reduce the number of nodes (related to area) and reduce the logic depth (related to delay) (MISHCHENKO, 2006). The next step is to map the circuit representation using logic gates, usually given by a technology standard cell library, known as technology mapping. After a covering step, several

optimizations are performed in order to: (1) meet the design constraints, such as delay; and (2) make cell area and power consumption as low as possible. The last step is the test logic insertion.

As already been stated, logic synthesis is a very complex task. The necessity of having a reasonable solution within time-to-market led to several heuristics, generating sub-optimal results, and left room for improvements. Finding optimal solutions may be feasible only for small circuits. In order to improve the QoR, an additional step was proposed, after the logic synthesis process. This extra phase, known as remapping or resynthesis (DE MICHELI, 1994; KUNZ, 1997), performs local transformations at the gate level (netlist) in order to improve the cost function of interest, such as cell area and power consumption.

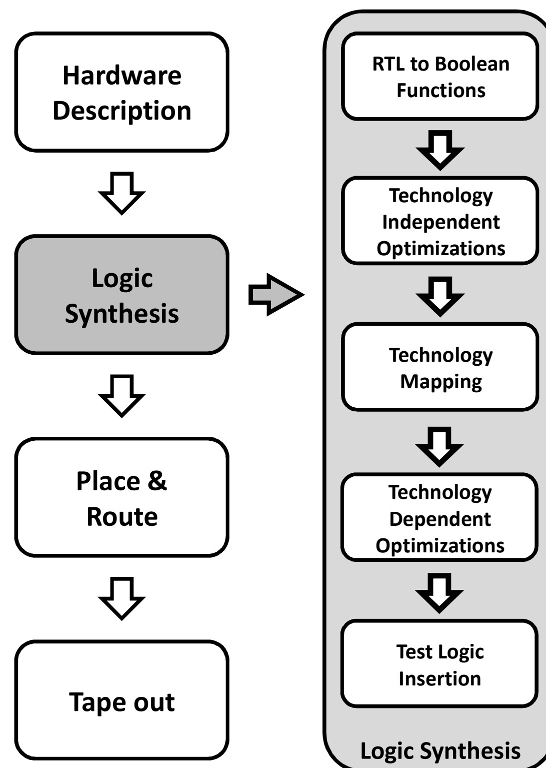


Figure 1.2: Logic synthesis in the standard cell design flow.

1.2 Motivation

Before the EDA industry and the scaling down of the transistor size, the development of ICs was very straightforward. There was a transistor network, which should be handmade and drawn in engineering papers, and all the process was understood and made by the development team. When the EDA industry began, enabling the development of larger circuits, along with the scaling down of the transistor size, numerous challenges appeared. Nowadays, besides the designed circuit working in the performance defined and having the smallest cell area and power consumption as possible, there are other concerns such as manufacturability, routing congestion, interconnection delay, leakage power, aging effects, radiation effects, lithography issues, and so on.

As the challenges arised, the EDA developers had to improve their tools in order to handle these new bottlenecks. The tools are being used and changed in the past 25

years, passing from several different generations. Every new problem tackled by the tool certainly made its code harder to read and harder to change. There is a rumor that the core of the world's most used commercial logic synthesis tool, is a black box that will probably never be changed again. Notice that, if new problems emerge or if new techniques to solve the current problems are discovered, there are basically two ways to incorporate them. An approach could be redoing all the code to perform all logic synthesis process, incorporating the new ideas and tackling the new problems. However, it is important to understand that a tool of more than 25 years of success will hardly be substituted by a new tool in any design house (DH). Also, the tool gives already very good results.

The motivation of this work is to provide an alternative approach: to perform the remapping of the gate level netlist given by a commercial tool, and to improve one or more cost functions of interest, using new strategies and tackling new problems that may arise.

1.3 Objectives

Remapping or resynthesis is not a new idea. Different approaches for resynthesis are already used to improve circuits after mapping, or even during the technology independent phase. In the work of (FIŠER, 2010), it is shown that the ABC tool (Berkeley Logic Synthesis and Verification Group, 2012) is able to achieve better results by performing iterative synthesis in random smaller parts of the circuit (sub-circuits) instead of performing synthesis in the circuit as a whole. It is important to notice that local optimizations can be applied in the results of different phases of logic synthesis: technology independent and technology dependent. From a technology independent point-of-view, local context can be extracted through windowing (MISHCHENKO, 2006) or by *k*-cut enumeration (PAN, 1994). The *k*-cuts approach can be considered a superior method to derive sub-circuits, since it is able to control the number of inputs of the Boolean functions present in a sub-circuit. For this reason, variations of *k*-cuts have been proposed, such as factor cuts (CHATTERJEE, 2006-b), priority cuts (MISHCHENKO, 2007) and *kl*-cuts (MARTINELLO JR., 2010). Interestingly, these advances in *k*-cut enumeration are strongly linked to the AIG data structure, and therefore to the technology independent phase of logic synthesis.

Approaches to local optimization of mapped circuits (i.e. remapping) adopt circuit partitioning techniques that do not consider the complexity of the Boolean functions in the resulting sub-circuits. For this reason, these remapping approaches lose local context, and need to investigate the surrounding environment (BENINI, 1998) to detect *controllability* and *observability don't cares* (SAVOJ, 1990), i.e. degrees of freedom. In the context of *k*-cuts in AIGs, the *observability don't cares* are incorporated in the sub-circuits due to the use of *k*-cut dominance.

The objective of this work is to bring the concept of *k*-cuts and *kl*-cuts from AIGs in order to be used on top of mapped netlists in a context of technology remapping. The *kl*-cut based remapping, when compared to the approaches proposed in the literature, introduces three important advantages: (1) to control the support cardinality; (2) all outputs affected by the cut inputs are found, making possible the logic sharing between outputs; and (3) a new concept of mapping flexibility through *polarity don't cares*, which is explained further in Section 4.5.

In order to validate the *kl*-cuts approach, an algorithm to enumerate *kl*-cuts on top of mapped circuits is proposed and implemented. Moreover, technology mapping techniques are implemented in order to improve a given cost function. In this work, two cost functions are considered: area and manufacturability. Furthermore, a static timing analysis (STA) tool is implemented in order to improve the cost functions of interest without affecting the performance obtained by the commercial tool. Hence, everything is put in a remapping flow and results are presented.

1.4 Thesis organization

The remaining of this thesis is organized as follows.

Chapter 2: *Technical background* – Provides the technical knowledge necessary to understand the concepts presented herein.

Chapter 3: *State-of-the-art* – Gives a broad vision of technology mapping and also provides the state of the art in important related topics to this thesis, such as Boolean factoring, Boolean matching and logic tree covering.

Chapter 4: *Cuts on mapped circuits* – Presents the first contribution of this thesis, the enumeration of *k*-cuts and *kl*-cuts on top of mapped circuits, and algorithms used for that.

Chapter 5: *KL-cut based remapping* – Describes the second contribution of this work, i.e. a complete flow, which is able to remap a circuit by extracting *kl*-cuts and replacing back, reducing circuit area.

Chapter 6: *Manufacturing cost function* – This is the third contribution of this work, which is a discussion about yield and lithography as a logic synthesis cost function, improving the number of good dies per wafer.

Chapter 7: *Results* – Presents and discusses the experimetal results. First, an analysis of the STA engine developed for this work is shown. Then, remapping results for IWLS 2005 benchmarks (IWLS 2005 benchmarks, 2012) are shown, reducing circuit area. The use of complex logic gates is also discussed. Finally, results of remapping using the manufacturing cost function are presented.

Chapter 8: *Conclusions and future work* – Outlines the conclusions and major contributions of this work, and also indicates future works.

2. TECHNICAL BACKGROUND

This chapter provides a review of technical concepts that are useful to the complete understanding of the remaining of this work. It comprises logic synthesis and logic circuit design topics, such as Boolean functions, expressions and networks, equivalence classes of logic functions, logic synthesis data structures, cuts on AIG, description of cell library, technology mapping, and static timing analysis. Readers with background knowledge on these topics can skip the following sections without loss in the understanding of the following chapters.

2.1 Boolean functions

The Boolean set is defined as $B = \{0, 1\}$, where 0 and 1 represent two well defined logic states, such as true (1) or false (0). An n-dimensional Boolean set B^n, is composed of 2^n distinct *Boolean vectors* with length n. For instance, $B^1 = \{0, 1\}$, $B^2 = \{00, 01, 10, 11\}$, $B^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$, $B^4 = \{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111\}$, and so on.

A *Boolean function f* is a function that relates every element in the n-dimensional Boolean set B^n (the function domain) into one element of the Boolean set B (the function image), such that $f : B^n \rightarrow B$. This means that each Boolean vector of length n is associated by a Boolean function to either 0 or 1. The Boolean function f has n *Boolean variables*. A Boolean variable can assume any value of B, i.e. it can assume the values 0 or 1. A Boolean vector is also known as *variable assignment*, which means that each position in the Boolean vector represents a variable assigned either to 0 or 1. For instance, the Boolean vector 0000 has the four variables assigned to 0.

A very usual way to represent a Boolean function is a *truth table*, such as the tables of Figure 2.1. On the left side of each table, each row represents a Boolean vector and each column represents the corresponding Boolean variables. On the right side, the columns represent the function names and the rows represent the value assumed by the function for the corresponding Boolean vector.

There are several logical operations that can be done with Boolean variables in order to generate different Boolean functions. The three basic Boolean operations are: AND (f_1), OR (f_2) and NOT (f_3), as seen in Figure 2.1. The operations AND and OR are binary, meaning that they must be applied to at least two operands. The AND operation returns 1 only if all operands are 1, returning 0 otherwise, as seen in Figure 2.1a. The OR operation returns 0 only if all operands are 0, returning 1 otherwise, as seen in Figure 2.1b. The NOT operation is a unary operation, which means that it is applied to only one operand, and applies a negation of the operand: if the operand is 0, the NOT returns 1, and vice-versa, as seen in Figure 2.1c.

A	B	*f1*
0	0	0
0	1	0
1	0	0
1	1	1

(a)

A	B	*f2*
0	0	0
0	1	1
1	0	1
1	1	1

(b)

A	*f3*
0	1
1	0

(c)

Figure 2.1: Truth tables representing the (a) AND, (b) OR and (c) NOT logical operations.

2.2 Boolean expressions

Besides a truth table, a Boolean function can also be represented as a *Boolean expression*. In this case, the Boolean operators are represented as the following symbols: AND operator is represented as * or ·, the OR operator is represented as +, and the NOT operator is represented as ! or ¬. These operators are applied to the variables of the function in order to represent correctly its functionality. Each time a Boolean variable appears in a Boolean expression, it is counted as one literal. The implementation of a Boolean expression with fewer literals is preferred, since it will likely use less logic elements, as discussed in Section 2.3.

Although a Boolean expression represents exactly one Boolean function, a Boolean function can be represented by numerous Boolean expressions. For example, let's take a look on the function *f*, represented in the truth table of Figure 2.2. Extracting the Boolean vectors that evaluate the function to 1, and representing them as Boolean expressions in order to represent the correct functionality of *f*, the result is the Equation (2.1), a sum-of-products (SOP).

a	b	c	d	*f*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Figure 2.2: Truth table representing function *f*.

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

Similarly, extracting the Boolean vectors that evaluate the function to 0, and representing them as Boolean expressions in order to represent the correct functionality of *f*, the result is the Equation (2.2), a product-of-sums (POS).

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

These two representations are straightforward, since they simply use the representation of the Boolean vectors as expressions, applying the correct logic operations in order to represent correctly the Boolean function. However, the SOP and POS representations have several literals. In Equation (2.1) and Equation (2.2) there are 32 literals each. In order to reduce the number of literals, the first approaches were with two-level minimizations (COUDERT, 1994). The two-level minimizations of the SOP and POS representations are in Equations (2.3) and (2.4), respectively, with 14 literals.

$$f = (!a * !b * c) + (!a * !b * d) + (a * b * c) + (a * b * d) + (c * d) \tag{2.3}$$

$$f = (!a + b + c) * (!a + b + d) * (!b + a + c) * (!b + a + d) * (c + d) \tag{2.4}$$

Further optimizations can be applied in order to decrease even more the number of literals, through multi-level minimizations, also known as *Boolean factoring* (BRAYTON, 1987). More details about Boolean factoring can be seen in Section 3.3. For instance, the minimal literal count expression of the function *f* can be seen in Equation (2.5), with 8 literals.

$$f = !((!c * !d) + ((!c + !d) * (a + b) * (!a + !b))) \tag{2.5}$$

2.3 Boolean networks

Figure 2.3 shows the schematic representation of the basic Boolean operators. In order to represent a Boolean function through a *Boolean network* schematic, the symbols of Figure 2.3 can be used to represent its implementation. Notice that the circle in the output of the NOT operator representation in Figure 2.3a, also known as inverter, shows the negation of this operator. The Figure 2.3b shows the representation of an AND operator, and the Figure 2.3c shows the representation of an OR operator. The NAND operator is created by adding a circle in the output of an AND operator, generating the AND output negated. The NOR operator is created by adding a circle in the output of an OR operator, generating the OR output negated. The NAND and NOR operators can be seen in Figure 2.4 and Figure 2.5.

The Boolean network can be directly derived from the Boolean expression. This derivation is not straightforward from the truth table representation, for example. Also, the representation of a Boolean function as a Boolean network shows the importance of reducing the number of literals in the corresponding Boolean expression. Notice that the the minimized SOP expression of *f* (Equation (2.3), with 14 literals) produces a larger Boolean network than the minimum expression of *f* (Equation (2.5), with 8 literals), as seen in Figure 2.4 and Figure 2.5.

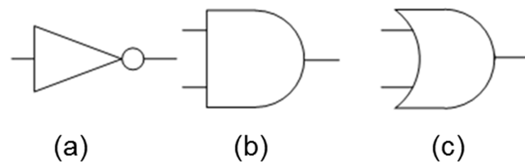


Figure 2.3: Schematic representation of (a) NOT, (b) AND and (c) OR operators.

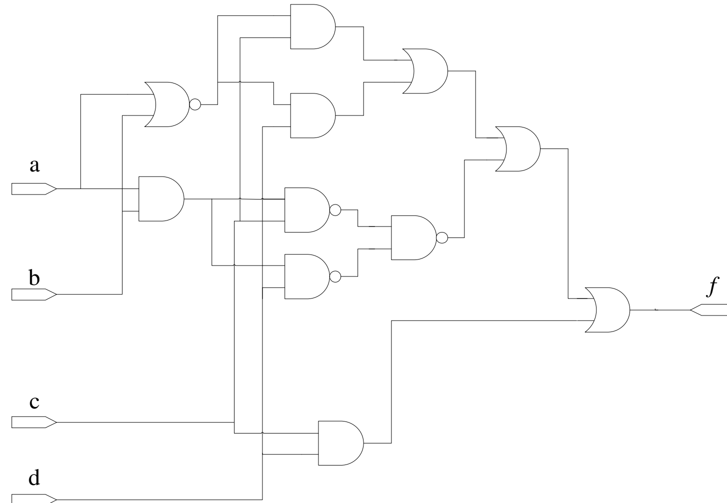


Figure 2.4: Boolean network mapped from Equation (2.3).

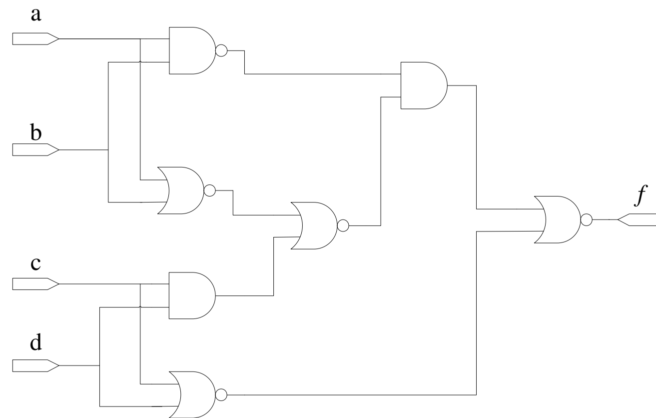


Figure 2.5: Boolean network mapped from Equation (2.5).

2.4 Equivalence of logic functions

Consider the following operations in logic functions: (P) permutation of one or more input variables; (Ni) negation of one or more input variables; and (No) negation of the function. If a function *g* is equivalent to function *h* by operation P, then the functions *g* and *h* are *P-equivalent*. In Figure 2.6, two Boolean networks are shown representing two distinct Boolean functions *pf1* and *pf2*, but in the same P-equivalence class (also at the same P-equivalence class that function *f* in Figure 2.5).

Also, if a function g_1 is equivalent to a function h_1 by performing the operations P and Ni, then the functions g_1 and h_1 are *NP-equivalent*. In Figure 2.7, two Boolean networks are shown representing two distinct Boolean functions *npf1* and *npf2*, but at the same NP-equivalence class (also at the same NP-equivalence class that function *f* in Figure 2.5, and functions *pf1* and *pf2* in Figure 2.6).

Finally, if a function g_{12} is equivalent to a function h_{12} by performing the operations P, No and Ni, then the functions g_{12} and h_{12} are *NPN-equivalent*. In Figure 2.8, two Boolean networks are shown representing two distinct Boolean functions *npnf1* and *npnf2*, but at the same NPN-equivalence class (also in the same NPN-equivalence class that function *f* in Figure 2.5, functions *pf1* and *pf2* in Figure 2.6, and functions *npf1* and *npf2* in Figure 2.7).

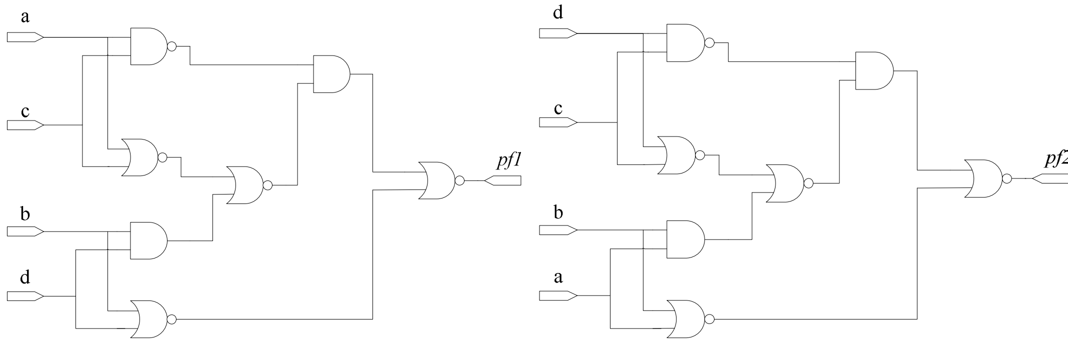


Figure 2.6: Boolean functions in the same P-equivalence class.

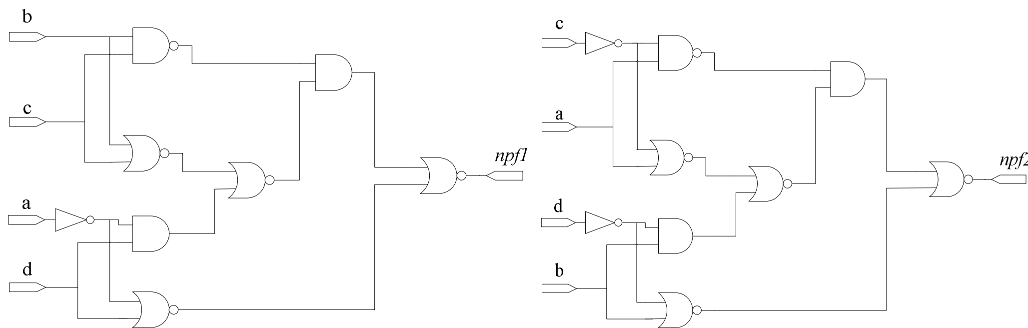


Figure 2.7: Boolean functions in the same NP-equivalence class.

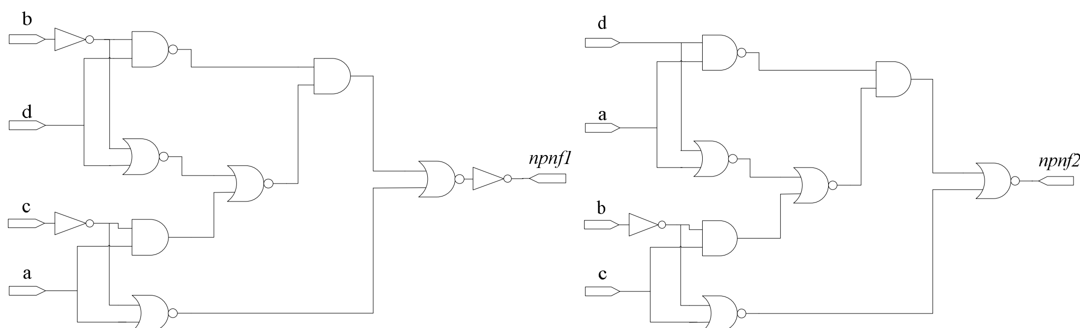


Figure 2.8: Boolean functions in the same NPN-equivalence class.

Notice that in all these equivalence classes, the core of the circuit remained the same. This characteristic is very useful in the technology mapping phase of logic synthesis. Also, these equivalence classes have a relationship, shown in the Venn diagram of Figure 2.9. For instance, there are a total of 65,536 Boolean functions with four variables, which can be classified in 3984 P-equivalence classes, 402 NP-equivalence classes or 222 NPN-equivalence classes (SASAO, 1999).

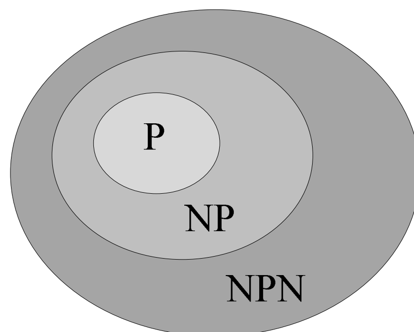


Figure 2.9: Venn diagram showing the relationship between Boolean functions equivalence classes: P, NP and NPN.

2.5 Data structures

A Boolean network can be represented through different data structures. Each data structure is more appropriate for different objectives and manipulations. Several data structures are used in this work, and the following subsections describe them.

2.5.1 Directed acyclic graphs

Graphs are widely used data structures in computer science, due to its efficient way of representing things and the also efficient algorithms created for graphs data structures. In order to represent a combinational circuit using a graph, it is necessary to guarantee the following conditions: (1) the edges must have a direction, i.e. the edges are directed; and (2) there are no cycles (cycles are prohibited in combinational circuits). These restrictions led to the use of directed acyclic graphs (DAGs) to represent circuits. DAGs can be used as a direct translation of a Boolean network into a graph data structure. An example of DAG can be seen in Figure 2.10.

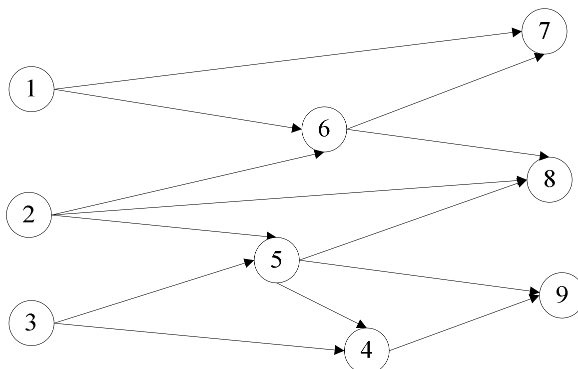


Figure 2.10: An example of directed acyclic graph.

2.5.2 And-inverter graph

An AND-inverter graph (AIG) is a specific type of a DAG, where each node has either zero incoming edges, the primary inputs (PIs), or two incoming edges, the AND nodes. Each edge can be negated or not. Some nodes are marked as primary outputs (POs). AIGs were created in order to perform fast transformations of circuits, since it is a very simple data structure (MISHCHENKO, 2006). An example of an AIG can be seen in Figure 2.11, where the nodes *a*, *b* and *c* are PIs, and the rest of the nodes are AND nodes. Also, the nodes *i*, *h*, *g* and *f* are marked as POs.

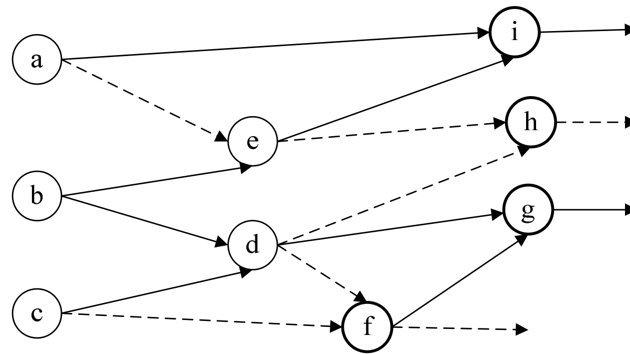


Figure 2.11: Example of AND-inverter graph.

2.5.3 Mapped circuit

A combinational mapped circuit *C* is a specific type of DAG with three types of nodes: the PI nodes, the *logic gate* nodes and the PO nodes. If a node of *C* has no incoming edges and one or more outgoing edges, it is a PI. If a node of *C* has up to *m* incoming edges, where *m* is an integer value such that $m \geq 1$, and one or more outgoing edges, it is a *logic gate* node. If a node of *C* has one incoming edge and no outgoing edges, it is a PO. An example of mapped circuit can be seen in Figure 2.12.

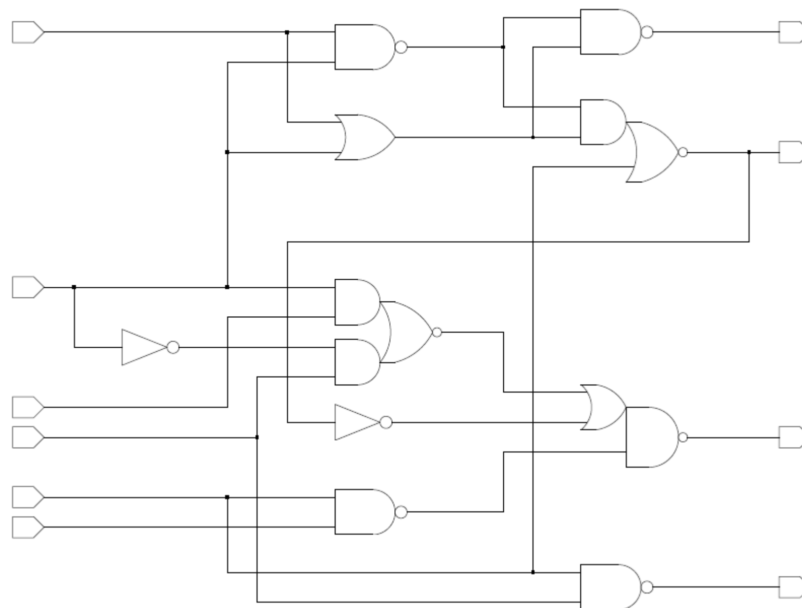


Figure 2.12: Example of mapped circuit.

2.5.4 Logic tree

A tree is a particular case of DAG in which the fanout of every node is equal to one. A logic tree is a specific type of tree in which the internal nodes are logic nodes, which represent logic functions such as AND and OR. A logic tree is also a direct translation of a Boolean expression into a data structure. An example of a logic tree is depicted in Figure 2.13.

It is computationally hard to map a DAG representing a circuit into logic gates, due to the several possibilities of mapping. The partitioning of DAG into several logic trees,

i.e. a forest of logic trees, reduces greatly the complexity of the DAG technology mapping, at the cost of reducing the solution space and producing a mapping result of lower quality. This approach will be explored in this work.

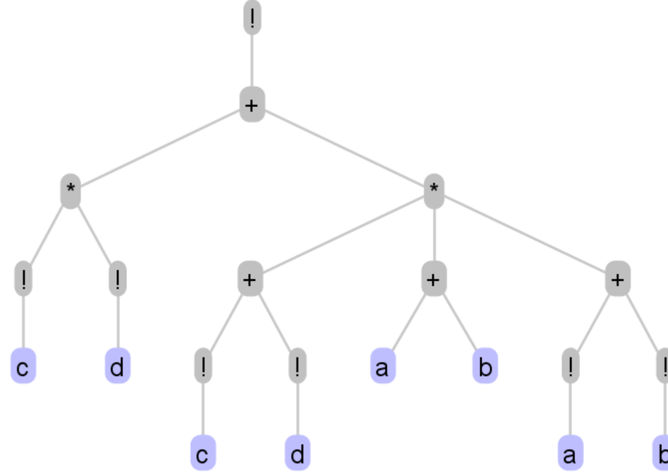


Figure 2.13: An example of a logic tree.

2.6 Cuts on AIGs

For scalability reasons, AIG is being used to represent circuits, since it is a very simple data structure (MISHCHENKO, 2006). Several transformations can be done with AIGs in order to reduce the number of nodes (area) or decrease the logic depth (delay). One way is to extract parts of an AIG through cuts and improve them locally. A *cut* of a node *n* in an AIG is a set of nodes *c* such that every path between a PI and *n* contains a node in *c*. A cut of *n* is irredundant if no subset of it is also a cut. This section describes *k*-cuts and *kl*-cuts on AIGs.

2.6.1 K-cuts

A *k*-feasible cut of an AIG $\mathcal{G}$ is an irredundant cut containing *k* or fewer inputs. Let A and B be two sets of cuts, and let the auxiliary operation ⋈ be the operation described in the Equation (2.6).

$$A \bowtie B \equiv \{a \cup b \mid a \in A, b \in B, | a \cup b| < k\} \tag{2.6}$$

Notice that the ⋈ operation is commutative, since the ∪ operation is also commutative. Let $\Phi_{\mathcal{K}}(n)\Phi_{\mathrm{K}}$ 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 be its inputs. Then, $\Phi_{\mathcal{K}}(n)\Phi_{\mathrm{K}}$ is defined recursively as described in Equation (2.7).

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

The ⋈ operation can easily remove the redundant cuts by comparing the cuts with one another. The *k*-cuts for all nodes of the AIG shown in the Figure 2.11 are described in Table 2.1.

Table 2.1: The k-cuts for all nodes of the AIG shown in Figure 2.11.

Node	k-cuts
a	{a}
b	{b}
c	{c}
d	{d}, {b, c}
e	{e}, {a, b}
f	{f}, {d, c}, {b, c}
g	{g}, {d, f}, {d, c}, {b, c}
h	{h}, {e, d}, {a, b, d}, {b, c, e}, {a, b, c}
i	{i}, {a, e}, {a, b}

2.6.2 KL-cuts

The k-cuts are an efficient way to represent a region of an AIG regarding one output generation. However, when it comes to multiple output regions, multiple k-cuts would be needed. The kl-cuts (MARTINELLO JR., 2010) make use of multiple outputs to overcome this issue.

A kl-cut defines a sub-graph $\mathcal{G}_{kl}$ of $\mathcal{G}$ which has no more than k inputs and no more than l outputs. It is represented as two sets of nodes $\{\mathcal{G}_k, \mathcal{G}_l\}$, being $\mathcal{G}_k$ the inputs set and $\mathcal{G}_l$ the outputs set. If a node n belongs to a path between $n_k \in \mathcal{G}_k$ and $n_l \in \mathcal{G}_l$, being $n \notin \mathcal{G}_k$, then n is contained in $\mathcal{G}_{kl}$. Notice that all nodes in $\mathcal{G}_l$ are contained in $\mathcal{G}_{kl}$. However, $\mathcal{G}_{kl}$ does not contain any node of $\mathcal{G}_k$. A kl-cut is said to be complete when all the following conditions are attained: (1) every path between a PI and a node $n_l \in \mathcal{G}_l$ contains a node in $\mathcal{G}_k$; (2) every path between a node contained in $\mathcal{G}_{kl}$ and a PO contains a node in $\mathcal{G}_l$; (3) no kl-cut defined by a subset of $\mathcal{G}_k$ and the same $\mathcal{G}_l$ is complete; and (4) no kl-cut defined by the same $\mathcal{G}_k$ and a subset of $\mathcal{G}_l$ is complete. Two examples of AIG covering using kl-cuts can be seen in Figure 2.14. If the kl-cuts with k=5 and l=3 (or simply 5-3-cuts) are computed, a resulting possible covering of this AIG is in Figure 2.14a. Another covering using 3-2 cuts is shown in Figure 2.14b. Further details on kl-cuts for AIGs can be seen in the work of Martinello (2010).

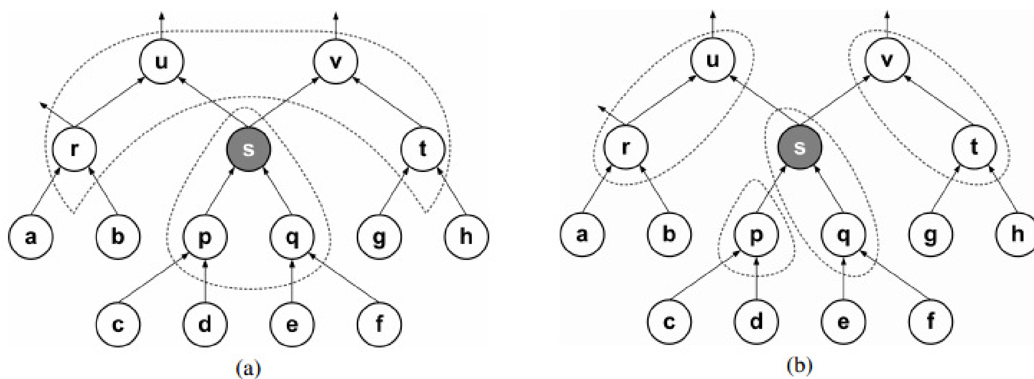


Figure 2.14 An AIG example illustrating covering using kl-cuts. Nodes *a, b, c, d, f, g* and *h* are primary inputs. Nodes *u* and *v* are primary outputs. (a) A covering using 5-3-cuts. (b) A covering using 3-2-cuts. (MARTINELLO, 2010)

2.7 Cell library

A logic gate, or a logic cell, is an element that performs a certain logic or memory function. The logic gates that implement Boolean functions are used to map the combinational part of logic circuits. Logic gates that perform memory function, such as latches and flip-flops, are used to create temporal barriers in order to generate sequential circuits, i.e. circuits that work in a well determined clock period.

A cell library is a finite set of logic gates. Traditionally, these logic gates are previously built and characterized through electrical simulations, resulting in well-defined cells to be used in the technology mapping. This approach is widely used and is known as *library-based* technology mapping, where the exact physical area, power consumption and delay of the cells are previously known. It is expected that a larger amount of logic gates would result in a higher QoR, since there are more options to reduce circuit area, for example. However, due to the high complexity of the technology mapping algorithms and the applied heuristics, a larger amount of logic gates does not necessarily improve the QoR.

A cell library is divided into several files. However, for logic synthesis two files are usually used: the *Liberty* file and the LEF file. The Liberty file is a standard text file that has the general library information, such as operating conditions (voltage, process and temperature), interconnections delay model, templates for timing and power tables, units for capacitance, voltage, time, etc. Also, the Liberty file has the information about all logic gates of the library, such as timing tables for each timing arc, logic function, input capacitances, power consumption tables, area, and so on. The library exchange format (LEF) file is a standard text file used to describe the geometrical shapes of the library cells layout, and also some geometrical restrictions of the IC manufacturing using this cell library. The standard cell flow has this name due to the standard height of the logic gates, and the information about the standard height is in the LEF file.

2.8 Technology mapping

Technology mapping is an important phase in the logic synthesis, which transforms a technology independent circuit description into a gate netlist of a technology library, i.e. a mapped circuit. It can be decomposed into three phases: decomposition, matching and covering.

Decomposition is the process that transforms the initial representation of the circuit into a simpler representation, more restricted, in order to make the process less computationally hard. In this step, it is applied structural transformations to the design representation, such as breaking the design graph into logic trees.

Once the circuit graph is computationally tractable, the matching step starts. The matching tries to find the parts of the graphs that can be implemented by a cell (or more than one cell) present in the library. In this step, the identification of Boolean functions in the same equivalence class is important, since a logic gate can implement different Boolean functions by performing the permutation of inputs, for instance.

Finally, the covering step chooses a subset of the match results in such a way that the entire circuit is covered, while optimizing one or more cost function such as area, delay and power. The result of the covering is a gate netlist that must correspond to the correct logic network received as input, i.e. all nodes of the input Boolean network must be covered.

The technology mapping is a phase that is crucial for a good QoR in the final circuit layout. The physical synthesis results depend directly on the technology mapping results. Also, it is important to notice that all steps in the technology mapping are important. Optimal algorithms for matching and covering deliver low quality results if the input design graph is not efficiently decomposed. This dependence in the previous structure is a problem known as *structural biasing* (CHATTERJEE, 2006-a).

2.9 Static Timing Analysis

Sequential digital circuits must be analyzed in order to check if there are no timing violations. This analysis is very important to determine if the design works correctly at the expected performance. Different approaches exist in order to check if there are no timing violations.

An approach to evaluate timing is through timing simulation, which is a method to perform timing analysis by testing all possible input vectors in the design. Timing simulation is a task that demands a lot of computational effort (BHASKER, 2009), since all different possibilities of inputs must be tested, i.e. a design with *n* inputs must check 2^n input vectors and check if any of these input vectors violates timing constraints. This means that a design with 100 inputs would have to test $1.26*10^{30}$ input vectors and propagate these vectors towards the design outputs, which is clearly a computational hard task.

An alternative approach to evaluate timing is the static timing analysis (STA). The static timing analysis is a fast way to analyze timing, considering only the worst case at each logic gate of the design. It is static, since it is the worst case, independent of the input vector. Also, since CMOS logic gates have different rise and fall characteristics, both cases must be considered. There is a small “penalty” in using this approach: STA is pessimist, since it considers only the worst cases. However, it is important to notice that it is better to have a timing check that guarantees that the design works, and in a reasonable time, since the timing check must be done several times in all phases of the design flow.

There are several delay models to determine the delay of a logic gate. The delay model most used in the standard cell design flow is the nonlinear delay model (NLDM). It determines the output transition time (t_{out}) and the output delay (t_d) of a logic gate through the input transition times (t_{in}) of the gate inputs and the capacitance load (c_L) at the gate output (BHASKER, 2009). Cells timing tables are read from Liberty files of the cell libraries, and using the t_{in} and the c_L, it is possible to obtain the correct t_{out} and t_d through bilinear interpolation. In Figure 2.15 shows which values are necessary to know in order to perform the bilinear interpolation and calculate the output transition time and cell delay, for a timing arc of an AND gate of 2 inputs, for the rise condition.

Any timing analysis tool is able to determine the circuit delays only due to the circuit itself. The environment conditions and restrictions must be informed to the timing analysis tool through the timing constraints in order to make such analysis more realistic. Timing constraints define, for example, the clock period and uncertainty, the input and output delays, input transition times and output loads. The standard format text file to inform the tool the timing constraints is the Synopsys design constraints (SDC) file.

In order to evaluate the worst case delay of a sequential circuit, which determines its performance (clock period), it can be applied the critical path method. In this method,

the delay is propagated from the inputs (or registers outputs) to the outputs (or registers inputs), considering only the worst case timing arc at each cell (for rise and fall). Clock uncertainty time and setup time must be added in registers endpoints. The input transitions of the evaluated cell are either: the input transition times defined in the SDC, in the case of an input pin, or the output transition times of the previous logic gates. The output load of the evaluated cell is the output load defined in the SDC, in the case of an output pin, added by all the input capacitances of all the cells the evaluated cell is driving. In the paths that involve input and output pins, input and output delays must be considered. The worst case delay is the largest delay from all endpoints (circuit outputs or register inputs).

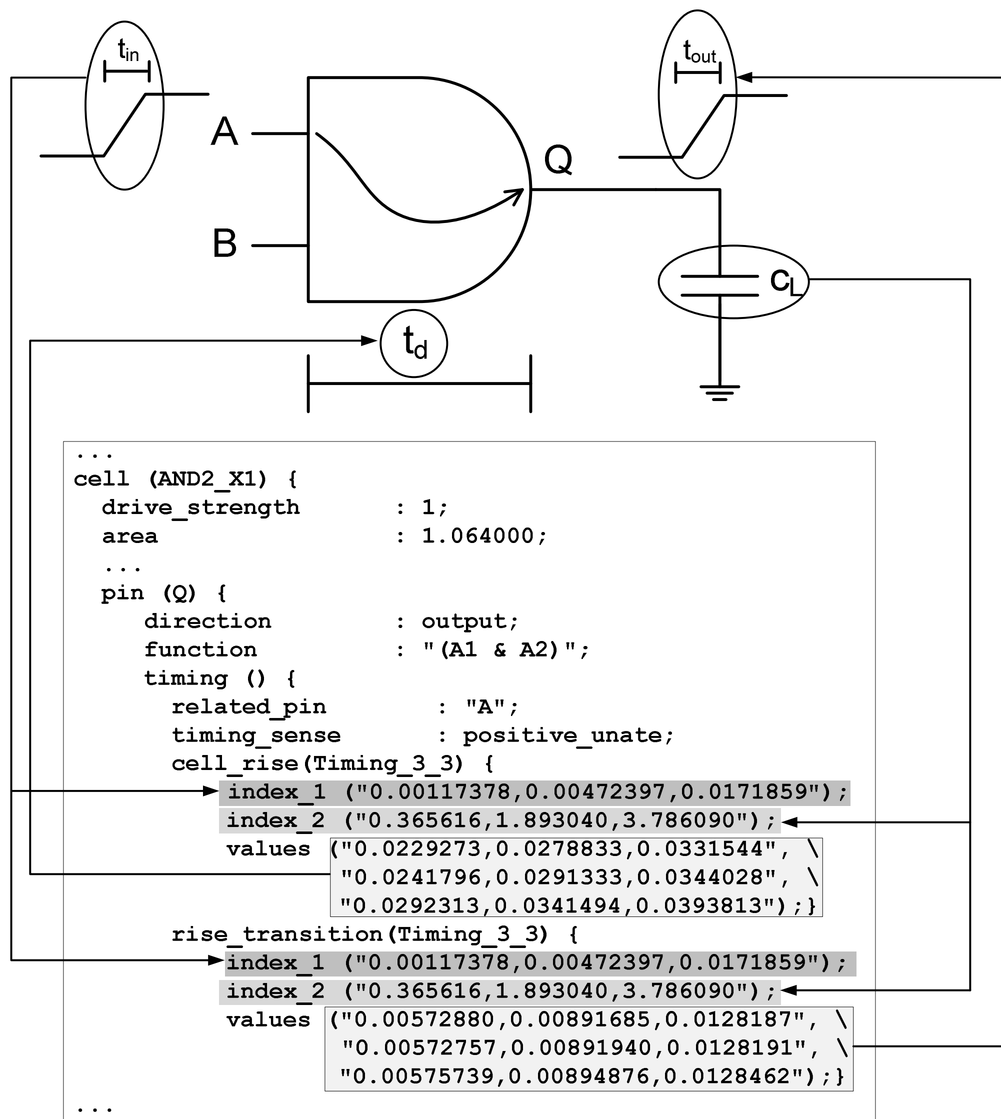


Figure 2.15: Process to obtain cell delay and output transition time through the NLDM, extracting the values from a Liberty file. It is necessary to know the input transition time and the output load, and perform a bilinear interpolation in the values read in the Liberty file.

3. STATE-OF-THE-ART

This chapter presents a broad vision about the technology mapping algorithms existent in the literature, showing their limitations and heuristics. Then, state-of-the-art algorithms on Boolean matching, Boolean factoring and logic tree mapping algorithms are shown, and as they were adapted to be used in this work.

3.1 Technology mapping

DAGON was the first technology mapping algorithm, which was proposed by Keutzer (1987). Keutzer noticed that the tasks performed by a software compiler were very similar to the tasks performed by the technology mapping. The pattern matching of sub-graphs of a circuit representation using library cells are very similar to the matching of sub-parts of a computer program using an instruction set of a computer processor. The subject graph circuit is a binary tree represented with a string. But the first technology mapping algorithm had some drawbacks. The structural matching approach performed by DAGON, and the representation of the circuit given in its input, restrict the search space performed by the mapping, affecting negatively the quality of the resulting mapped circuit. Another issue is that the algorithm requires all isomorphic matches stored in each node of the circuit tree representation, until the very end of the circuit covering. This led to a restriction of using cell libraries with a large number of cells, because the number of cells affects directly the number of pattern matchings found in each node.

The first method that used logic trees as subject graph was proposed by Detjens (1987). It has some similarities with DAGON, and some additions. It was the first work that proposed the use of a pair of inverters in every non-inverted net of the circuit representation. This approach increases the solution space, impacting positively in the quality of the resulting mapped circuit. However, the approach proposed by Detjens had to create several decompositions for each library cell available, increasing exponentially the amount of pattern matchings, and also limiting the use of larger cell libraries.

Mailhot (1993) improved the technology mapping algorithm by improving the comparison between the sub-trees and the cells of the library using ROBDDs (NARAYAN, 1997). Like the previous algorithms presented (KEUTZER, 1987; DETJENS, 1987), the proposed approach split the initial circuit DAG into a forest of trees and maps them individually. But since ROBDDs are a canonical form of representing circuits, the matches did not depend in the structure of the sub-trees, but in the Boolean function it represented. However, this Boolean comparison was computationally expensive, also limiting the size of the cell library used.

An approach to minimize the dependence in the initial graph representation was proposed by Lehman (1995), using a dynamic reorganization. The decomposition step

was integrated with the pattern matching by making sub-graphs functionally equivalent but structurally different associated, for each node of the circuit DAG. This approach increased the search space, but it is also impractical for large circuits since the graph size increases exponentially.

Stok (1999) proposed the algorithm *wavefront*, similar to Lehman's approach but with no scalability problems. The circuit was mapped directly in the DAG, using a delay model independent of the cell output load. In order to prevent the DAG of increasing exponentially by inserting different representations for each node, the steps of decomposition, pattern matching and covering are executed in parallel in a "window" of parameterizable logic depth. This "window" performs the mapping from the inputs to the outputs, and has better results and runtime if compared with its predecessors.

The state-of-the-art in library-based technology mapping approaches is still the work presented by Chatterjee (2006-a) It brings together several techniques used in logic synthesis integrated in order to benefit the technology mapping. The pattern matching is performed by Boolean matching and the data structure used is the AIG. This algorithm and its improvements are all incorporated in the academic logic synthesis tool ABC (Berkeley Logic Synthesis and Verification Group, 2012).

By performing this holistic view of some of the most important technology mapping approaches presented in the literature, which the commercial tools are strongly based, some limitations were observed. Two are important limitations that this work tries to take advantage: (1) the scalability issues due to the use of a large cell library, and (2) the scalability issues of performing mappings with a large solution space in large circuits.

3.2 Boolean matching

The pattern matching is an important step in the technology mapping, which can be perfomed using a structural analysis, i.e. comparing two structures if they are the same, as it can be done in the circuits of the Figure 2.6. This comparison is computationally easy and several technology mapping algorithms took advantage of this to have faster results. However, a logic function can have numerous structures to represent it, and therefore a structural matching obtains very limited results.

Another way to perform pattern matching is through a functional (Boolean) analysis of the two functions. This comparison is computationally harder than a structural matching, but it is much more powerful, since it does not depend on the structural implementation of the circuit. This work uses Boolean matching in order to check if two functions are in the same P-equivalence class.

A well-known Boolean matching algorithm for any type of function was proposed by Hinsberger (1998). The method is based on the definition of a canonical representative function $R[f]$ for each equivalence class f. The matching between two functions happen when the representative function calculated is the same: $R[f_1] = R[f_2]$.

A function can be defined through a truth table, which can be represented as a bit string. The method proposed by Hinsberger (1998) uses the largest number that the given function can represent permutating the input variables as the representative function. For example, Figure 3.1 shows an exhaustive approach for finding this representative function for a function of three variables. In the Figure 3.1, the values in the parenthesis show in the top the variable selected (1, 2 or 3), and the position in the function in which the function was set (1, 2 or 3). Notice that the input variables must

obey a very strict ordering. Also, notice that the leftmost value (10101000) represents the truth table of the input function, since it places the variable 1 in the position 1, the variable 2 in the position 2 and the variable 3 in the position 3. In this case, the representative function is 1110000 since it is the largest value found.

The computation of which branch gives the largest value can be done earlier, choosing the next branch at each tree level. This is done in the Figure 3.2, where the search space was reduced by deciding the largest function value earlier.

Another reduction that can be done in order to reduce further the amount of computation is to only check the variables in the same *symmetry* class. Two variables *a* and *b* of a function *f* are in the same symmetry class when they can be exchanged without changing the resulting function: $f(a,b) = f(b,a)$. In the Figure 3.2, the variables 2 and 3 are in the same symmetry class for this function, and therefore only one of the branches should be checked: (1,2) or (1,3).

In the work of Martinello (2010), it is proposed an extension of this Boolean matching algorithm in order to match multiple-output functions, where it is defined a PP-equivalence class. This extension is very important to group *kl*-cuts in the same equivalence class and improve the remapping flow runtime, with no affect in the quality of the resulting circuit.

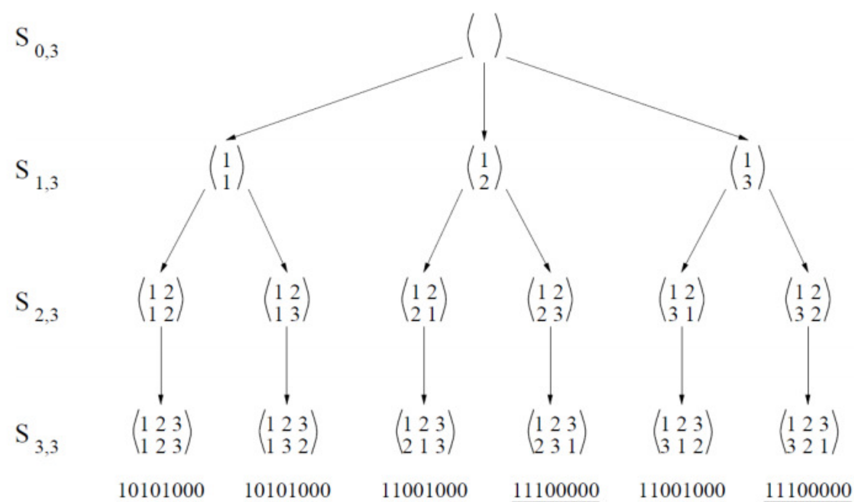


Figure 3.1 Exaustive approach for computing $R[f]_P$ (HINSBERGER; KOLLA, 1988)

3.3 Boolean factoring

Factoring is the process of deriving a parenthesized algebraic equation representing a given logic function (BRAYTON, 1987). Factoring algorithms can be classified into algebraic and Boolean.

The algebraic factoring has its basis in polynomial division, pretending that the Boolean variables behave like real numbers. The basic concept is that, given the functions *f* and *p*, the algorithm tries to find functions *q* and *r* such that $f = p \cdot q + r$. The function *p* is called a divisor of *f* if *r* is not null, and a factor if *r* is null. Some relationships can be used to simplify the results during factoring. The relationships used in algebraic factoring are shown in the left column of Table 3.1. Algebraic factoring is usually very fast, but commonly the results are far from optimal.

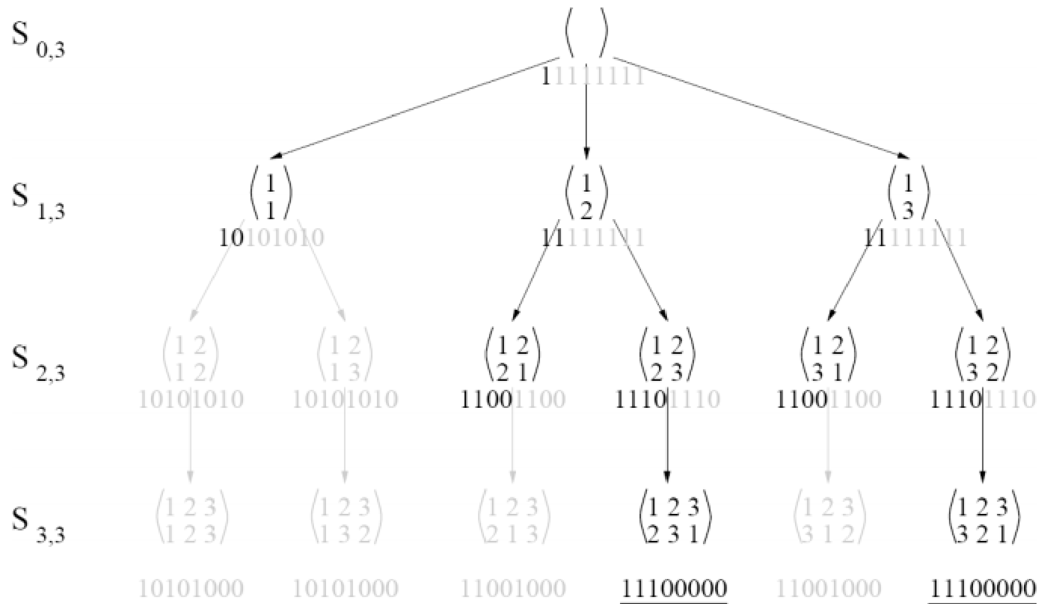


Figure 3.2 Reducing search space by cutting non-maximal branches (HINSBERGER; KOLLA, 1998)

Table 3.1 Relationships allowed in multi-level logic factoring

Relationships allowed in algebraic and Boolean factoring	Relationships allowed only for Boolean factoring
a•b= b•a	a+a’ = 1
a+b= b+a	a•a’ = 0
a•(b•c) = (a•b)•c	a•a= a
a+(b+c) = (a+b)+c	a+a= a
a•(b+c) = a•b+ a•c	a+1 = 1
a•1 = a	a+(b•c) = (a+b)•(a+c)
a•0 = 0	
a+0 = a	

On the other hand, Boolean factoring algorithms achieve better results, but they can be very time and memory consuming, since it is an NP-complete task. The state-of-the-art Boolean factoring algorithm was presented in the work of Martins (2012), using the functional composition (FC) paradigm. The FC exact algorithm uses dynamic programming and a bottom-up approach to find minimum literal logic expressions. But the exact approach is impractical to be used in a remapping flow, since it uses a lot of time and memory. In the same work presented by Martins (2012), several heuristics were added to this algorithm in order to make it faster and have still obtain very good results. The FC heuristic algorithm has comparable runtime to previous works and results in logic expressions with smaller literal count. An example of how the FC heuristic algorithm works is shown in the Figure 3.3: from the function variables (selected in the correct polarity) the functions are associated until the target function is found.

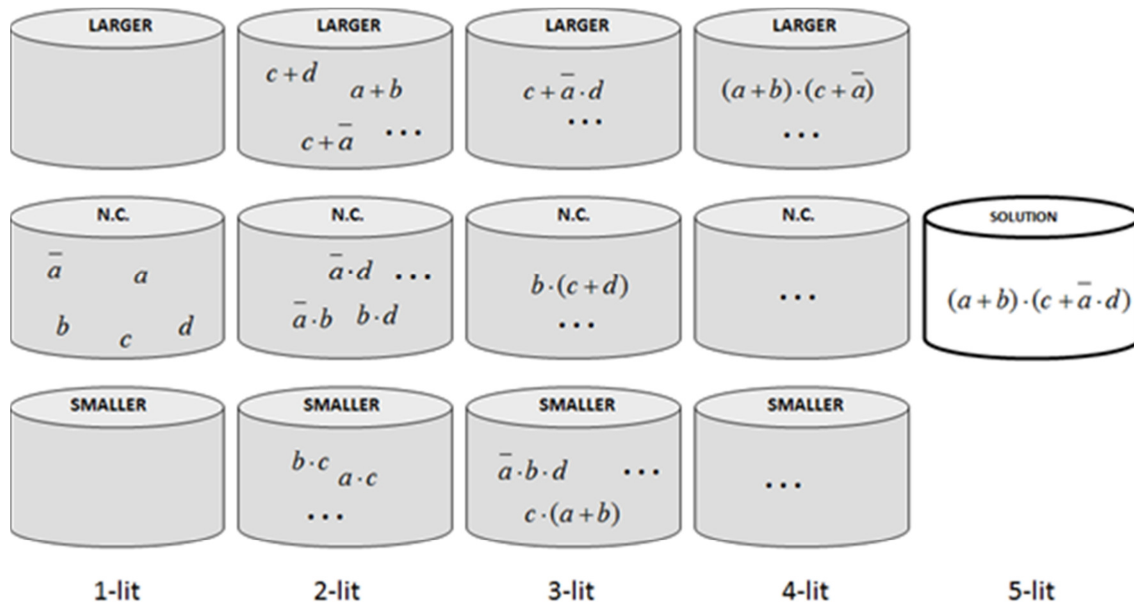


Figure 3.3 Example of Boolean factoring using Functional Composition. (MARTINS, 2012)

Besides the FC heuristic algorithm, this work uses a modified version of it, regarding the use of the XOR operator (MARTINS, 2012), and also logic sharing. In order to use logic sharing, two approaches were used. A simpler approach was to use the other outputs functions (all outputs but the current output being factorized) as input to the FC algorithm as one literal functions. Another approach was to identify partial logic functions that were used by more than one output and use a dynamic cost to these intermediate functions (one literal or more), according to the effective use of them.

3.4 Logic tree mapping

The output of the FC algorithm is a set of logic expressions. This logic expressions can be represented as logic trees and mapped, resulting in a mapped circuit. This logic tree mapping is not straightforward and requires several transformations to obtain a good logic tree covering.

A very good logic tree mapping algorithm was proposed by Correia (2004). It performs several transformations in the logic tree (DeMorgan's theorem, grouping of equivalente nodes, decompositions of a logic node) in order to have different options of tree covering, which may lead to different results. The algorithm proposed by Correia (2004) was intended to be used in a library-free flow, so the actual implementation of the standard cells is not known during the mapping. The cost calculation step, as seen in the Figure 3.4, is performed using the number of series/parallel (s,p) transistors at each node. In this case, 2 transistors in series or 2 transistors in parallel was used as the limit to represent a logic gate, and the tree is then divided in sub-trees with this maximum cost. By performing iteratively this algorithm, the resulting cover and mapped circuit can be seen in the Figure 3.5.

Several modifications of this algorithm were performed in order to adapt this tree mapping algorithm to a technology dependent context. The first change was the cost: the number of transistors is not a good measure for a technology dependent mapper. So,

a cost is given in the tree mapper input (area, power, etc.) and the cost is calculated at each logic tree node using the information of the library cells. The pair of inverters approach was also used, increasing the search space. The resulting tree mapping algorithm performs a Boolean matching for every node of the tree, saving all minimal implementations for all nodes of the tree, and returning a tree mapped with the cell library given with the minimal cost desired, disconsidering logic depth and timing.

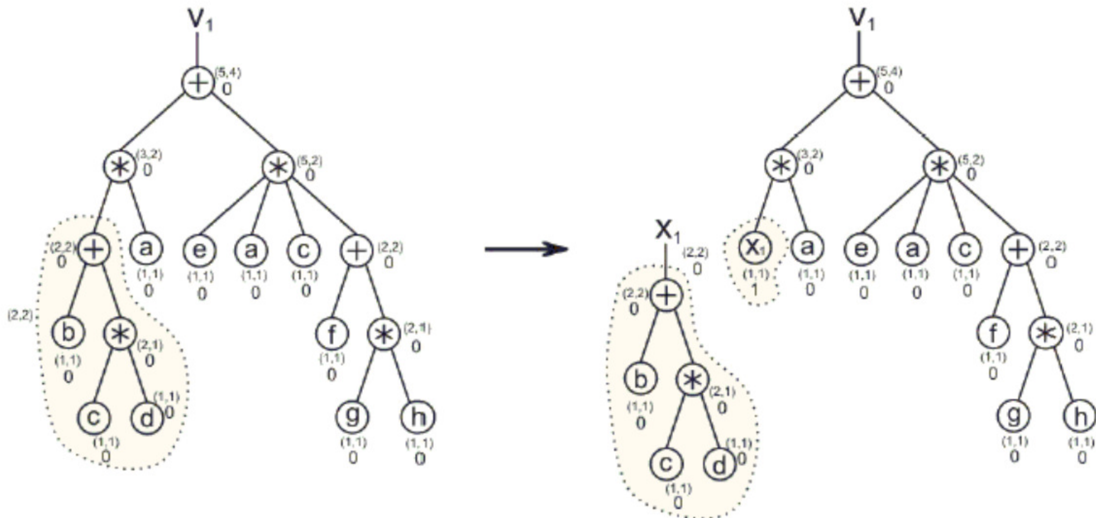


Figure 3.4 Cost calculation and the first cut of the tree removed. (CORREIA, 2004)

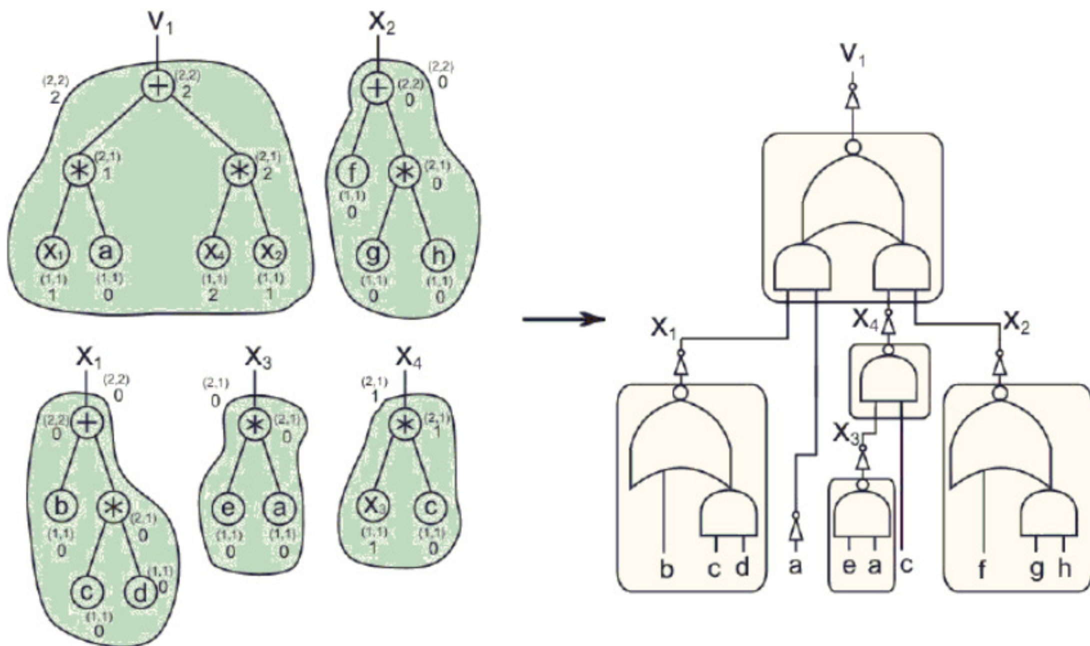


Figure 3.5 Example of logic tree covering (before inverter minimization). (CORREIA, 2004)

4. CUTS ON MAPPED CIRCUITS

This chapter presents the first contribution of this work, which is the enumeration of cuts on top of mapped circuits. The use of *kl*-cuts is inspired in the work of Martinello (2010), which introduced such idea for AIGs, a technology independent data structure. This work tries to improve the results of the logic synthesis process, which results in a technology dependent mapped circuit, by performing local optimizations. These local optimizations are performed inside the sub-circuits found through the *kl*-cut enumeration algorithm.

4.1 Differences between AIGs and mapped circuits

And-inverter graphs and mapped circuits are two different types of DAGs, which can represent a logic circuit. The main differences between AIG and mapped circuit descriptions are: (1) the number of incoming edges on the nodes; and (2) the existence of inverters and buffers instead of simply negated or direct edges. In order to extend the concept of *k*-cuts to mapped circuits, it is necessary to handle these differences.

4.1.1 Number of nodes inputs

The nodes on AIGs simply perform the AND operation, and are limited to two inputs (MISHCHENKO, 2006). This is because the AIG was created to be a very simple and scalable data structure, in order to perform complex and computational hard logic minimization techniques. Other logic operations, besides AND operation of two inputs, need inverters (which are in the edges of the AIG) and AND nodes arranged in a way to perform the logic operation desired. Any logic function (of any inputs) can be implemented using the AND logic operation and inversions.

The nodes in mapped circuits can perform any logic function, from one input (inverters and buffers) to *m* inputs, where *m* is the integer number representing the number of inputs of the logic gate with the largest number of inputs in the library. This data structure is not simply a logic circuit representation, but also a logic circuit implementation, i.e. a data structure of a circuit that can be built as is.

4.1.2 Inverters as nodes or edges

In AIGs, the negation is performed in the edges instead of using specific nodes for that. This representation simplifies the methods of logic minimization based on AIGs, making them more scalable and simple. In real world mapped circuits, negations on the nets do not exist. In order to perform signal invertions, an inverter is necessary, i.e. inversions are perfomed in nodes of the graph in mapped circuits, instead of edges in AIGs. Besides inverters, there are also buffers, which are one input logic gates that are used to decrease the delay to load larger capacitances, without changing the logic function.

4.2 K-cuts

The k-cuts on top of a mapped circuit $\mathcal{C}$ must take into account the higher amount of inputs at internal nodes and one input logic gates (inverters and buffers), which are not present in AIGs. Let $\Phi_{\mathcal{K}}(n)\Phi_{\mathrm{K}}$ to be the set of k-cuts of $n \in \mathcal{C}$, and if n is a logic gate node, let $n_1, \dots, n_g$ to be its inputs, where g is an integer value representing the number of inputs of n such that $1 \leq g \leq m$. By using the same operation ⋈ described in Section 2.6.1, $\Phi_{\mathcal{K}}(n)\Phi_{\mathrm{K}}$ is defined recursively as seen in Equation (4.1). Figure 4.1 shows a combinational circuit example. By enumerating the k-cuts with k=5 for the example in Figure 4.1, the values given in Table 4.1 are obtained. Notice that *wire0* does not appear in any k-cut, since it is the output of a cell with a single input.

$$\Phi_{\mathcal{K}}(n) \equiv \begin{cases} \{n\}, & : n \text{ is a } PI \\ \Phi_{\mathcal{K}}(n_1), & : g = 1 \\ \{n\} \cup \{\Phi_{\mathcal{K}}(n_1) \bowtie \dots \bowtie \Phi_{\mathcal{K}}(n_g)\} & : otherwise \end{cases} \tag{4.1}$$

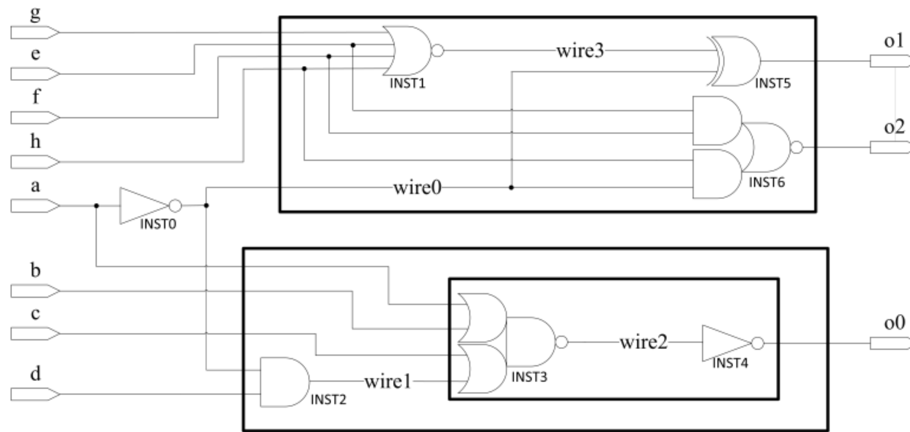


Figure 4.1 Combinational circuit example to demonstrate k-cuts and kl-cuts computation.

Table 4.1 All k-cuts with k=5 for all nodes of the combinational circuit example of Figure 4.1

Node	*k*-cuts
a	{a}
b	{b}
c	{c}
d	{d}
e	{e}
f	{f}
g	{g}
h	{h}
wire0	{a}
wire1	{wire1}, {d, a}
wire2	{a, b, c, d}, {a, b, c, wire1}, {wire2}
wire3	{wire3}, {e, f, g, h}
o0	{a, b, c, d}, {a, b, c, wire1}, {wire2}
o1	{o1}, {a, wire3}, {a, e, f, g, h}
o2	{o2}, {a, e, f, h}
all nodes	{a, b, c, d}, {d, a}, {a, wire3}, {e, f, g, h}, {a, b, c, wire1}, {a, e, f, g, h}, {a, e, f, h}

4.3 KL-cuts

The *kl*-cuts on top of AIGs are sub-parts extracted in order to have different options to cover the AIG functionality afterwards, mapping to a FPGA for example. The objective of enumerating *kl*-cuts on top of mapped circuits is to improve a cost function of the cuts and then replace them in the original circuit. Hence, *kl*-cuts formed by only one cell or with *k* equals to one (e.g. inverter or buffer chains) are not considered. Furthermore, in order to identify all possible shared logic for a given set of inputs, the *l* is defined as unbounded, not limiting the number of outputs and keeping track of all outputs that depend on the same set of variables.

The *kl*-cuts introduce important advantages, such as: (1) the control of the support cardinality, and (2) the possibility of performing logic sharing between outputs, since all outputs affected by the *kl*-cut inputs are found. In order to better understand the advantages of using *kl*-cuts, consider a simple and-inverter-graph (AIG) example, as it is shown in Figure 4.2. If it is desired to create a sub-circuit in which the node 14 is the output with up to 4 inputs, a search backwards would find that the node 14 depends on the nodes {4, 5, 11, 8} and stop, due to the five input barrier made by nodes {4, 5, 6, 7, 8}. That is a limitation in the sub-circuit search in (KUNZ, 1997; BENINI, 1998; KRAVETS, 2004; MISHCHENKO, 2006; FIŠER, 2010), which will not find that node 14 depends directly on nodes {a, b, c, d}. By performing *k*-cuts enumeration, it is guaranteed that node 14 depends directly on nodes {a, b, c, d}.

This is an important feature of *k*-cuts, in which the *kl*-cuts are based. Nevertheless, a *k*-cut generates only one output, i.e. a *k*-cut does not cover all outputs it affects. Notice the mapped circuit in Figure 4.3, which is structurally similar to the AIG of Figure 4.2. Consider the cover of the *k*-cut {a, b, c, d} generating the logic at the output of the logic gate 14. In order to cover logic gate 13, while respecting the cover for logic gate 14, it is necessary to duplicate part of the logic that is common to both logic gates. By identifying all outputs that the input nodes of a *k*-cut affects, a *kl*-cut is found, making possible to remap it locally and replace it in the circuit netlist, keeping the logic equivalency and efficiency. It is important to notice that *kl*-cuts provide a complete input-output interface for a sub-circuit substitution. Additionally, *kl*-cuts minimize the support of the Boolean functions inside the cut. By minimizing the support, it is possible to apply aggressively Boolean minimization techniques that would not scale for larger circuits.

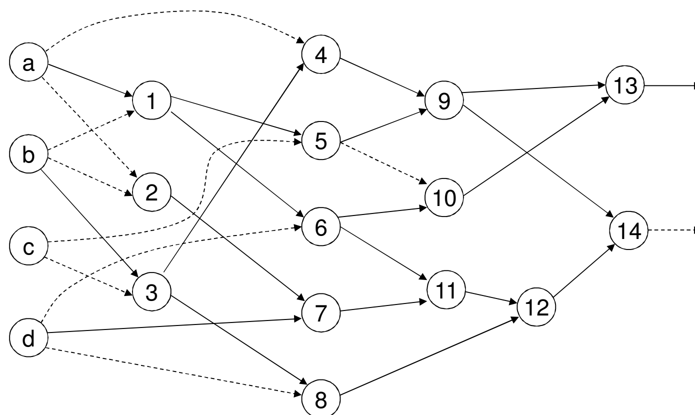


Figure 4.2 And-inverter graph (AIG) representing a circuit.

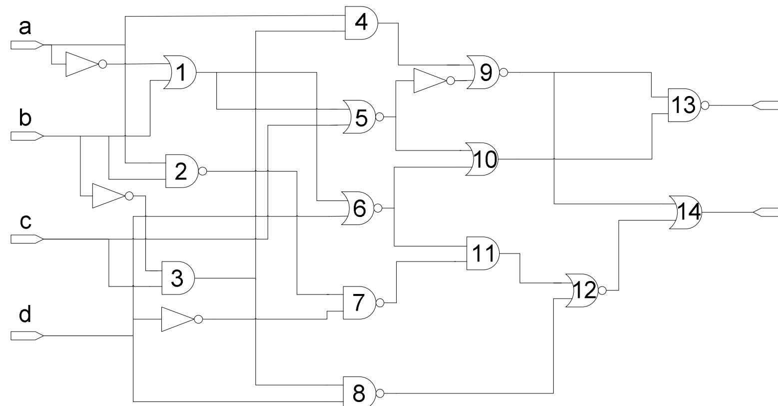


Figure 4.3 Mapped circuit structurally similar to the AIG of Figure 4.2.

4.4 Enumeration algorithm

A pseudo-code for *kl*-cuts enumeration on top of a mapped circuit is shown in Figure 4.4. The algorithm receives a *k* limit and a mapped circuit. If the design has sequential elements, it is necessary to treat these sequential elements as PIs and POs to the combinational logic. The algorithm starts by enumerating all *k*-cuts for all nodes of the circuit. All *k*-cuts of all nodes are grouped, such that each *k*-cut can generate one *kl*-cut. For example, the line *all nodes* of Table 4.1 represents the *kcuts* (line 2) for the circuit in Figure 4.1.

The function *addInsts()* traverses the circuit from each *kcut* input (line 3) to the outputs direction, storing the logic gate instances, the outputs generated and all logic functions. Each logic gate node is checked by the function *KCutsOK()*, which returns true if the node has at least one *k*-cut formed only by *kcut* inputs. If *KCutsOK()* returns false, or the node is a PO, the node checked is a *kl*-cut output. Notice that a *kl*-cut of a circuit can be exchanged by another, since all signals which are affected by the cut are taken into account. Thus, the use of *kl*-cuts in remapping is justified. In the circuit example of Figure 4.1, three *kl*-cuts are found, which are shown with rectangles around the instances contained in each *kl*-cut.

```
01. compute_klcuts(k, circuit) {
02.   kcuts = compute_kcuts(circuit, k)
03.   for each kcut in kcuts do {
04.     insts <- ø
05.     outputs <- ø
06.     for each node in kcut do {
07.       addInsts(node, insts, outputs)
08.     }
09.     klcut = createKLcut(kcut, insts, outputs)
10.     klcuts.add(klcut)
11.   }
12.   return klcuts
13. }
```

Figure 4.4 Pseudo-code for kl-cuts enumeration on top of a mapped circuit.

4.5 Polarity *don't cares*

After identifying a *kl*-cut instances, inputs, and outputs, a further search is performed on the inputs, identifying inverters and buffers. All the inverters and buffers that are used only to drive the *kl*-cut can be encapsulated. The inverters and buffers that drive not only the cut, but also other parts of the circuit, must not be encapsulated. Notice that there is an inverter (*INST0*) in Figure 4.1 not encapsulated by the *kl*-cuts found, since it is used in distinct parts of the circuit.

If these inverters or buffers are encapsulated, there is a duplication of these cells during remapping (as in the following example) or an overlap of *kl*-cuts (explained further in Section 5.2.7.1). During the search on the inputs, the inverters that are not encapsulated can be used to generate a mapping flexibility: the *polarity don't cares*. A similar approach can be done in the flip-flops that generate both polarities of a signal.

A *kl*-cut found in a commercial benchmark is shown in Figure 4.5. Notice that the *kl*-cut has two *polarity don't cares* (i9=!i0 and i12=!i1). Using the *polarity don't care* information in the remapping tool, there is a reduction of 20% in area of the *kl*-cut. If this information is not used, the reduction is of only 6%. The circuits obtained with and without *polarity don't care* information are shown in Figure 4.6 and Figure 4.7, respectively. Notice that there is an addition of two inverters (*INST6* and *INST7*) in Figure 4.7.

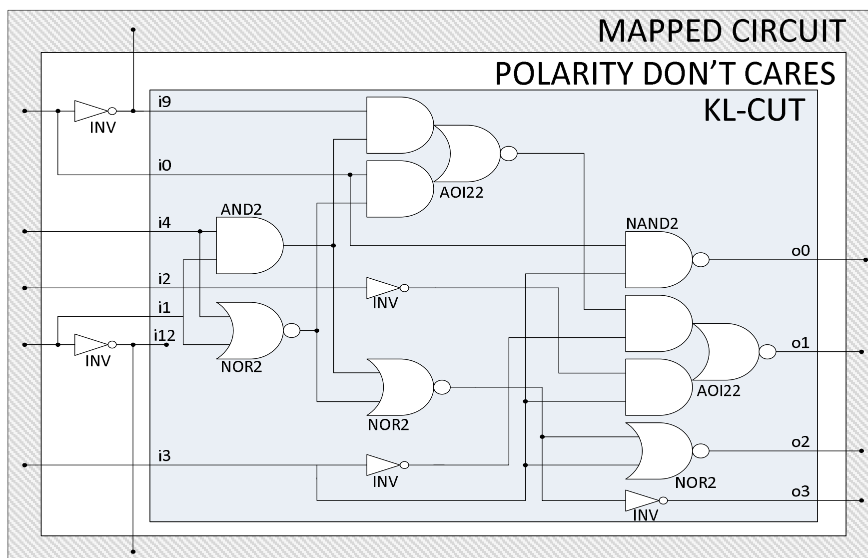


Figure 4.5 Example of *kl*-cut found in a commercial benchmark.

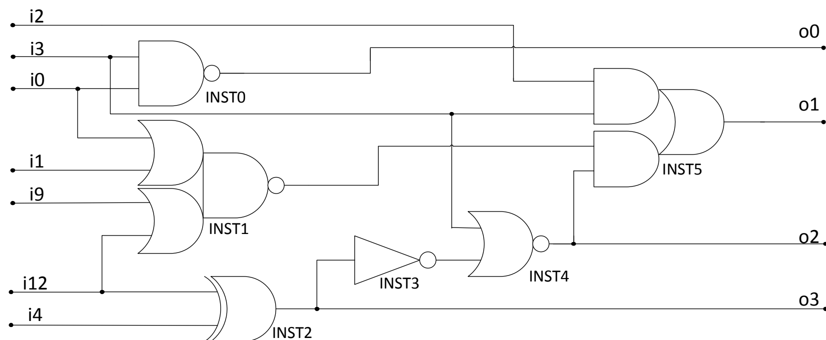


Figure 4.6 Example of Figure 4.3 remapped with *polarity don't cares* information.

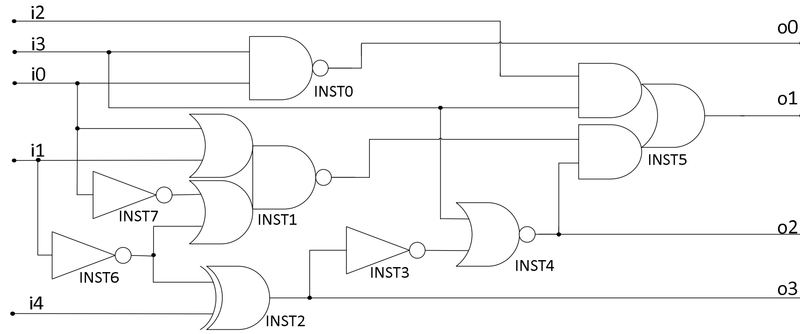


Figure 4.7 Example of Figure 4.3 remapped without *polarity don't cares* information.

4.6 Degrees of freedom

In the work proposed by Benini (1998), an extra search in the neighborhood of the partitioned circuit is performed in order to identify the degrees of freedom (SAVOJ, 1990). In Figure 4.8, for example, an extra search in the neighborhood is able to notice that, if '*i1'* and '*i4'* are equals to '0', the outputs of the "logic gate netlist" *don't care*, which are known as *observability don't cares*. By enumerating the *kl*-cuts with (at least) *k*=4 in Figure 4.8, the outputs *o1* and *o2* depend directly on the inputs *i1*, *i2*, *i3* and *i4*, and this *don't care* information is self-contained in the *kl*-cut. This search in the neighborhood is necessary in the work proposed by Benini (1998) because the logic gate netlist is created randomly, by selecting a node and adding more nodes to it. Since the *kl*-cuts are enumerated from the inputs to the outputs, the *observability don't cares* due to the *kl*-cuts inputs will be *satisfiability don't cares*, and then can be removed during the resynthesis.

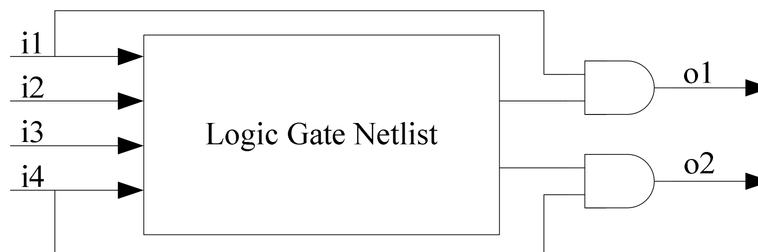


Figure 4.8 Logic circuit example.

4.7 Conclusion

This chapter presented a comparison of *k*-cuts and *kl*-cuts performed on top of mapped circuits as opposed to computing *k*-cuts and *kl*-cuts on top of AIG representations. The main differences lie on (1) the number of inputs for the 2-input AND nodes used on AIGs and the nodes of a gate netlist which may have several inputs, and (2) the existence of explicit inverters and buffers, appearing as nodes, in the netlist compared to the use of negated or direct edges used in the AIG. Moreover, algorithms to enumerate *k*-cuts and *kl*-cuts on top of a netlist representation were proposed and implemented.

5. KL-CUT BASED REMAPPING

This chapter presents the main contribution of this work, which is the use of cuts on top of mapped circuits in order to perform iterative remapping. A complete operational flow is presented, with details about every step of the flow.

5.1 Remapping using KL-cuts

The advantages of the current approach are linked to the use of *kl*-cuts to enumerate the sub-circuits. Previous remapping approaches of mapped circuits adopt circuit partitioning techniques that do not consider the complexity of the Boolean functions in the resulting sub-circuits (DEY, 1990; KUNZ, 1997; BENINI, 1998). For this reason, these remapping approaches lose local context, and need to investigate the surrounding environment to detect *observability don't cares* (BENINI, 1998). By using *k*-cuts, the *observability don't cares* are incorporated in the sub-circuits due to the *k*-cut characteristic of dominance (PAN, 1998). In this work, the concepts of *k*-cuts and *kl*-cuts are used on top of mapped circuits in the context of technology remapping. When compared to the approaches proposed in (KUNZ, 1997; BENINI, 1998; KRAVETS, 2005; MISHCHENKO, 2006; FIŠER, 2010), *kl*-cuts introduce important advantages, such as (1) the control of the support cardinality, and (2) the possibility to perform logic sharing between outputs, since all outputs affected by the *kl*-cut inputs are found.

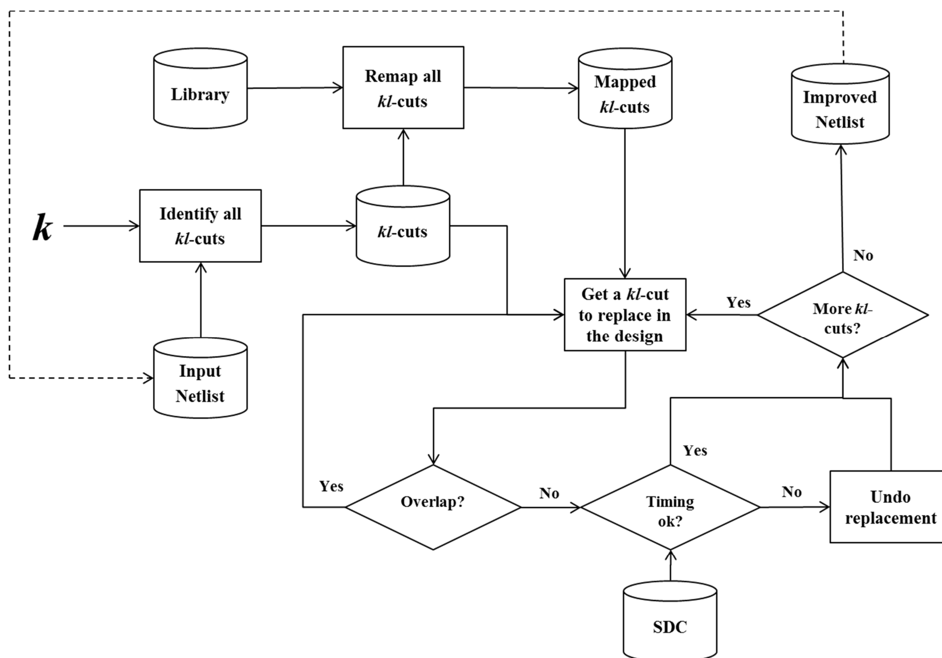


Figure 5.1 Proposed *kl*-cut remapping flow.

5.2 KL-cut remapping flow

The proposed flow is shown in Figure 5.1. In order to perform the remapping, the following inputs are necessary: the gate-level netlist of the design and the liberty files (library) used to map it. If it is desired to have a more precise timing analysis, an SDC file must be read as well. If no SDC is provided, the remapping flow will improve the cost function regardless of timing constraints. After reading the necessary files, the remapping is performed. A common flow is shown in Figure 5.2, using the tool developed, named KLever2. It is necessary to define the maximum size of the *kl*-cut inputs *k* (the functions support), which is 6 in this case. The number of iterations desired *n* is also necessary. For *n*=0, the tool performs remapping iterations while it is still possible to improve the cost function. For *n* different of 1, the improved netlist is used again in the flow, in order to perform the next iteration, as illustrated by the dashed line in Figure 5.1. In this example, an area reduction of 13.6% was found after 3 iterations for c432 benchmark (IWLS benchmarks 2005, 2012).

```
C:\Windows\system32\cmd.exe

C:\test>java -jar klever2.jar -f remap.script
Klever2 - KL-cuts for Verilog
>help read_liberty
usage: read_liberty [-e] -f <filename> [-l <lithography_file>] -o <name>
Read a Liberty files and returns a Library Map
 -e                       Find extra cells
 -f <filename>            Liberty file name
 -l <lithography_file>    Text file with lithography information
 -o <name>                Library name
>read_liberty -f base_library.lib -o lib
>
>help read_verilog
usage: read_verilog -l <library> -o <design> -v <verilog>
Read a Verilog file and returns Circuit Description
 -l <library>    Library
 -o <design>     Design Description name
 -v <verilog>    Verilog file name
>read_verilog -v c432.netlist.v -l lib -o design
>
>help read_sdc
usage: read_sdc -d <design> -f <sdc_file> -o <sdc_name>
Reads an SDC file
 -d <design>      Design description input
 -f <sdc_file>    SDC file name
 -o <sdc_name>    SDC name
>read_sdc -f c432.sdc -d design -o sdc
>
>help iterative_remap
usage: iterative_remap -d <design> -k <k_inputs> -l <library> [-m] -n
        <n_tries> -o <design> [-s <sdc>] [-t] [-y]
Does all the process of remapping
 -d <design>      Design description input
 -k <k_inputs>    Max size of Cut Inputs
 -l <library>     Library name
 -m               Map only not golden cut
 -n <n_tries>     Number of remapping iterations (0 for brute force)
 -o <design>      Output design description
 -s <sdc>         SDC for the design
 -t               Map tree cuts first
 -y               Remap targetting yield
>iterative_remap -l lib -d design -s sdc -k 6 -n 0 -o remapped_design -m
Info: Iteration number 1 with KL-Cuts with K=6 Heuristic targetting area - Current: 121.482
Info: Iteration number 2 with KL-Cuts with K=6 Heuristic targetting area - Current: 113.526
Info: Iteration number 3 with KL-Cuts with K=6 Heuristic targetting area - Current: 104.95799999
>
>help stats
usage: stats [-o <filename>]
Show statistics
 -o <filename>    Save statistics information to a file
>stats
Design:
        design
                c432    I:36    O:7     W:143   C:107   A:121.482       CA:121.482      SA:0.000

        remapped_design
                c432    I:36    O:7     W:129   C:93    A:104.958       CA:104.958      SA:0.000
Library:
        lib                     266 cells with 49 functions.
SDC:
        sdc
>
>help write_verilog
usage: write_verilog -d <design> -l <library> -v <verilog>
Write Verilog file from Circuit Description
 -d <design>     Design description
 -l <library>    Library name
 -v <verilog>    Verilog output file name
>write_verilog -d remapped_design -l lib -v c432.remapped.netlist.v
>
>quit
```

Figure 5.2 Example of a complete remapping script.

The remapping starts by enumerating all *kl*-cuts and remapping them. Then, the remapped cuts are sorted from the highest to the lowest gain according to the cost function desired, performing a greedy selection. Notice that the mapped *kl*-cuts selected to be replaced in the design are only the ones that improve the cost function. The *kl*-cuts can be overlapped (details in subsubsection 5.2.7.1), so it is checked if there is no overlapping before replacing them back in the netlist. If there is no overlapping, the *kl*-cut is replaced and the timing is checked. If the timing remains acceptable, the replacement is approved and the *kl*-cuts replacement continues until there are no more *kl*-cuts to replace.

5.2.1 Liberty parser and data structure

The Liberty file is a standard cell library file format used in the standard cell flow. The Liberty file contains information about the library cells, such as: the cell name, the input and output pins names, the logic function performed by each output, the timing and power tables, the cell area, the inputs capacitances, etc. The power tables have been neglected in the parser developed to this work, since the power information was not studied in this first version, i.e. only area and timing was considered. Figure 5.3 shows the information read from an AND2 gate of a Liberty file, which is possible to get with the command *print_library_cells*.

```
.SUBCKT AND2_X1 A1 A2 Q VDD VSS
*.PININFO A1:I A2:I Q:O VDD:P VSS:G
*.COMBINATIONAL_LOGIC_GATE
*.AREA_VALUE 1.224
*.CAP_VALUES RISE: A1:9.08E-4 A2:9.98E-4 FALL: A1:9.32E-4 A2:0.00103
*.EQN Q=(A1 * A2)
*.TIMING_TABLES
* OUTPUT_PIN: Q
*    INPUT_PIN: A1
*    UNATENESS: POSITIVE_UNATE
*    INPUT_NET_TRANSITION_INDEX: [0.0010, 0.02, 0.04]
*    OUTPUT_TOTAL_CAPACITANCE_INDEX: [2.80653E-4, 0.00407697, 0.00815394]
*    CELL_RISE: [(0.0226,0.04134,0.060053)
                 (0.031938,0.050609,0.069501)
                 (0.041727,0.060403,0.079288)]
*    CELL_FALL: [(0.019064,0.031426,0.042973)
                 (0.029097,0.04149,0.053071)
                 (0.039568,0.052127,0.063734]
*    RISE_TRANSITION: [(0.008929,0.028669,0.05085)
                       (0.008986,0.028663,0.050975)
                       (0.009931,0.028854,0.051089]
*    FALL_TRANSITION: [(0.006406,0.017923,0.030511)
                       (0.006481,0.017936,0.030482)
                       (0.007601,0.018293,0.030684)]
*    INPUT_PIN: A2
*    UNATENESS: POSITIVE_UNATE
*    INPUT_NET_TRANSITION_INDEX: [0.0010, 0.02, 0.04]
*    OUTPUT_TOTAL_CAPACITANCE_INDEX: [2.80653E-4, 0.00407697, 0.00815394]
*    CELL_RISE: [(0.025413,0.044382,0.063154)
                 (0.034863,0.053786,0.072692)
                 (0.044417,0.063269,0.082138)]
*    CELL_FALL: [(0.019785,0.031108,0.041521)
                 (0.029581,0.040946,0.051377)
                 (0.040131,0.051612,0.062045)]
*    RISE_TRANSITION: [(0.009158,0.028734,0.050867)
                       (0.009174,0.028725,0.050995)
                       (0.009716,0.028841,0.050939)]
*    FALL_TRANSITION: [(0.006249,0.016571,0.027764)
                       (0.006313,0.016564,0.027796)
                       (0.00732,0.0169,0.027867)]
.ENDS
```

Figure 5.3 Cell information read from a Liberty file.

Figure 5.4 shows an example of a library with four combinational cells. It was implemented two ways to access a cell in the library: (1) using the cell name, which is important while parsing a Verilog file, for example; and (2) using the cell function P-signature, which is important during technology mapping.

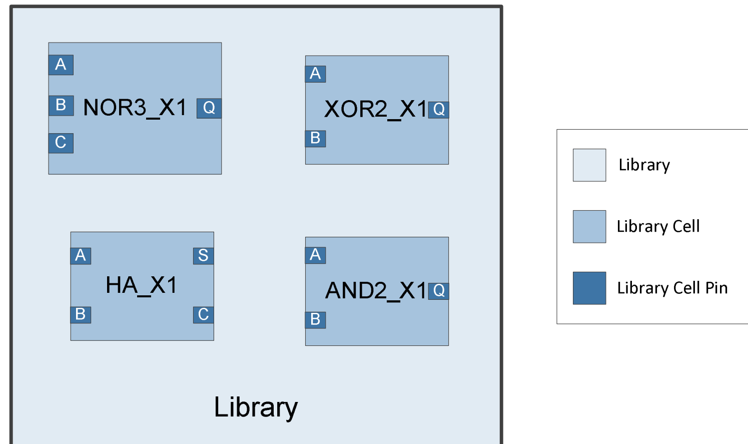


Figure 5.4 Example of a library with four different logic gates.

5.2.2 Verilog parser and data structure

The gate-level netlist of a given design can be described in different hardware description languages. One of the most used in digital design is the Verilog language, which was used in this work. An example of the Verilog format accepted in the tool is illustrated in Figure 5.5. It is expected that the input Verilog has only one module, with the header formatted in Verilog 1993 format: *module <module_name> (<list_of_terminals>);*. Then, the inputs, outputs and wires are described, followed by the cells instantiations. Assignments are also accepted. When the Verilog file is read, a circuit data structure is created, similar to the example in Figure 5.6.

```
module Design(i0, i1, i2, i3, o0, o1, o2);
       input i0, i1, i2, i3;
       output o0, o1, o2;
       wire x1, x2, x3;
       NOR3_X1 U2(
              .A(i1),
              .B(i2),
              .C(i3),
              .Q(x3));
       AND2_X1 U4(
              .A(i0),
              .B(i1),
              .Q(x1));
       XOR2_X1 U1(
              .A(i1),
              .B(i2),
              .Q(x2));
       XOR2_X1 U2(
              .A(x2),
              .B(x3),
              .Q(o2));
       HA_X1 U2(
              .A(x1),
              .B(x3),
              .S(o0),
              .C(o1));
endmodule
```

Figure 5.5 Example of structural Verilog.

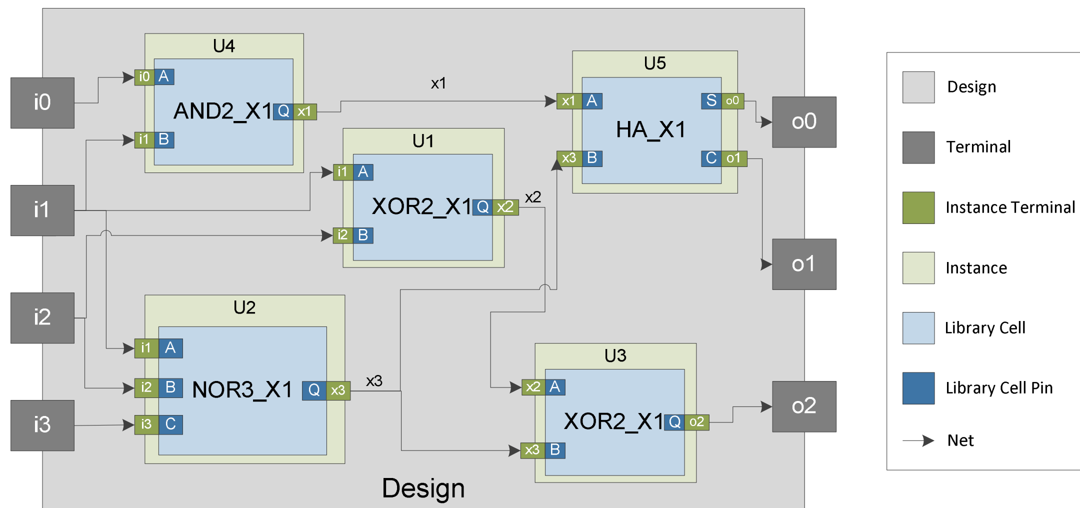


Figure 5.6 Example of the circuit data structure.

The circuit data structure has the design name (Verilog module name), a list of the inputs names, a list of the outputs names, a map to the cell instances, a map to the driving nodes, and the circuit area (which is the sum of the area of all instantiated cells). The instance data structure has the instance name, a map with the inputs names, a map with the outputs names, a pointer to the library cell, and the timing information, which is updated when the STA is performed. For example, the maps of the instance U3 in Figure 5.6 are the following. Inputs map: (A=x2, B=x3); outputs map (Q=o2). The node data structure has the net name, the instance name and the pin name, or the terminal name (input or output). There are two types of nodes, the source and destination. The source nodes have a list of pointers to the fanout nodes which they drive, and the destination nodes have a pointer to the fanin node which drives them.

5.2.3 SDC parser and data structure

A timing analysis tool is able to determine timing only due to the circuit itself. The environment conditions and restrictions must be informed to the timing analysis tool through the timing constraints, in order to the make the analysis more realistic. Timing constraints define, for example, the clock period and uncertainty, the input and output delays, input transition times and output loads. The SDC commands currently read in the proposed STA engine are described in Table 5.1.

5.2.4 KL-cut enumeration and data structure

The *kl*-cut enumeration is performed as described in the algorithm of Section 4.4. By defining the *k* (*k*>1), which represents the maximum number of inputs, the *k*-cuts are found and each *k*-cut generates a *kl*-cut. It is very important to limit the number of inputs of the sub-circuits found, since it is a limiting variable in Boolean minimization algorithms, increasing time exponentially with the number of inputs. An example of a *kl*-cut found is in Figure 5.7. This data structure can be obtained by the command *get_cuts* followed by the command *print_cuts*.

The *kl*-cut data structure has the cut name, which is unique; a pointer to the circuit data structure; a map with the inputs names and a map with the outputs names, such as the “identifiers” row in Figure 5.7; a list of pointers to the instances of the cut; a list of the intermediate wires; the *kl*-cut area, which is the sum of the instances’ area; a map with the logic expressions of the outputs; and the *polarity don’t cares* available.

Table 5.1 Commands currently read in the STA engine developed.

Command name	Description
set_input_delay	Defines the delay necessary to add in a path starting at a circuit input, necessary in input-to-register and input-to-output paths.
set_output_delay	Defines the delay necessary to add in a path starting at a circuit output, necessary in input-to-output and register-to-output paths.
set_clock_transition	Defines the transition time of a clock signal, necessary to define the delay and the setup time of the registers.
set_input_transition	Defines the transition time of an input signal, necessary to define the delay and output transition time of the cells driven by input terminals.
set_load	Defines the capacitance load at an output, necessary to define the delay and output transition time of the cells that drive output terminals.
set_clock_uncertainty	Defines the error margin of a clock period. This value must be added to all paths, in order to guarantee that timing will be attained. Clock skew and clock jitter are the values usually used.
create_clock	Defines the clock name, pin and period.
set_logic_zero	Defines an input as zero, enabling constant propagation and logic minization.
set_logic_one	Defines an input as one, enabling constant propagation and logic minization.
set_false_path	Defines a path as false for timing analysis.

```
.SUBCKT E4 i0 i1 i2 i3 i4 i5 i9 o0 o1 o2 o3 VDD VSS
*.PININFO i0:I i1:I i2:I i3:I i4:I i5:I i9:I o0:O o1:O o2:O o3:O VDD:P VSS:G
*.IDENTIFIERS i0:N82 i1:N76 i2:n106 i3:n105 i4:n103 i5:n23 i9:N223 o0:n1 o1:n36 o2:n93 o3:n78
*.INSTANCES U142:CLKINV_X1 U140:INV_X0D5 U183:NOR2_X1 U110:OAI21_X0D5 U108:NAND3_X1 U155:NAND2_X1
*.AREA_VALUE 5.508
*.DESIGN_NAME c432
*.INTERMEDIATE_WIRES n21 n45
*.POLARITY_DONT_CARES i5 i9
*.EQN i5=!i0
*.EQN i9=!i4
*.EQN o0=!((!((i0 * i4) + (i0 * !i1)) + !i2) + !i3)
*.EQN o1=(!i3 + (!((i0 * i4) + (i0 * !i1)) + !i2))
*.EQN o2=(!i1 * i0)
*.EQN o3=((!i1 * i0) + i4)
.ENDS
```

Figure 5.7 Example of *kl*-cut data structure representation.

5.2.5 KL-cut P-group

A design can have a lot of *kl*-cuts. For instance, a design with 10,000 logic gates can have easily 100,000 *kl*-cuts. However, lots of these cuts have similar logic, which can be grouped in P-classes, reducing the amount of cuts from 5,000 to 20,000. In the current implementation, the remapping does not consider local timing information while the technology mapping, so the *kl*-cuts can be grouped without any lose of quality.

In order to decrease the amount of remappings, which is the bottleneck of the current flow, an extension of the multiple output PP-signature introduced by Martinello (2010) has been implemented. This extension was necessary to also consider *polarity don't cares*. The idea is the following: the inputs have an order defined by the PP-signature, and *polarity don't cares* are assigned to the inputs using this order. Consequently, two *kl*-cuts are in the same PP-class if they have the same PP-signature, and if the same variables in the PP-signature order have *polarity don't cares* available.

Another technique used in order to decrease the runtime was to keep a database of the best implementation cuts, which was called *golden cuts*. By using the parameter *–m*

in the *iterative_remap* command, the tool keeps the best implementation of each P-class to be used in the following iterations so reducing runtime.

5.2.6 KL-cut remapping approach

The remapping is performed trying to improve a cost function. The cost function primarily used is area, which is a historically common cost function in the logic synthesis and technology mapping. The remapping starts in the Liberty reading, where the lowest cost cells are selected for each P-signature. Using area as cost function, the cells with drive strength equals to 1 (X1 cells) were selected, since they have the smallest area in the cell library.

During the remapping flow, all *kl*-cuts are remapped. The logic expressions of each *kl*-cut is processed by a factoring algorithm, based on Martins (2012) Functional Composition (FC) paradigm, which reduces the amount of literals in each logic expression. The “circuit DAG” in this case is minimized to a set of logic expressions. Notice that the mapping for minimum area in a DAG is NP-complete, but if the DAG is partitioned into trees, each tree can be mapped with minimum area with linear time.

Currently, there are different ways to factorize the equations performed: factoring each output expression independently (single output); use the output functions as inputs to other outputs (multiple output); and identify portions of logic that can be used by more than one output (partial multiple output). The FC algorithm used can perform factorization using the basic operators, resulting expressions with {AND, OR, INV} operators, and also using the *XOR* operator, which results in expressions with {AND, OR, XOR, INV} operators. Therefore, each *kl*-cut can have its logic expressions in up to six sets of expressions: single output, multiple output, partial multiple output, with and without the XOR operator. It is important to notice that the factoring results are technology independent, which means that the best set of expressions may differ for different technologies. For instance, a technology with complex cells with area comparable to simple cells can obtain better results by factoring the outputs expressions independently.

After the factoring step, each set of logic expressions factorized is passed to a logic tree mapper. Each logic expression of the set is mapped independently, using the *polarity don’t care* information, i.e. the circuit was divided in a forest of trees. The tree mapping algorithm is based on the Correia’s (2004) algorithm, trying to improve a cost function. By connecting the logic tree mapper results, a *kl*-cut circuit is generated for each of the logic expression factorized sets. The following example shows how this remapping approach is able to reduce area of *kl*-cuts, and consequently of the circuit.

Consider the following *kl*-cut example, found in one of the benchmarks examinated. The *kl*-cut inputs are *i0*, *i1*, *i2*, *i3*, *i4*, *i9*, *i12*; and the outputs are *o0*, *o1*, *o2*, *o3*, as seen in Figure 5.8. There are two *polarity don’t cares* available: *i9=!i0*, and *i12=!i1*. The area of this sub-circuit is 9.792 um², as found in the mapped circuit benchmark. By performing the proposed factoring, four different sets of logic expressions were found. The original logic expressions of the *kl*-cut outputs and the different factorized expressions are on Table 5.2.

Notice that the multiple output with XOR expressions have less literals than the multiple output results. Nevertheless, the best area result is obtained by the circuit mapped with multiple output expressions, since the logic tree mapping algorithm was able to use the SPI7F165 complex gate (which logic expression is *(a*b)+c*(d*e+f*g)*).

This result is also interesting since the *xor* operation inside *o1* expression *(!i0*i1)+(i0*!i1)* is performed in series/parallel format, because of the *polarity don't cares* available, *(i9*i1)+(i0*i12)*, as seen in Figure 5.9.

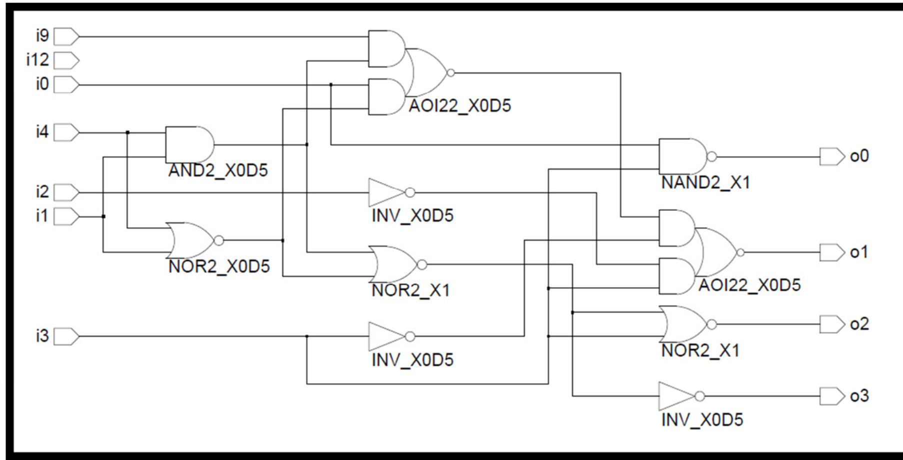


Figure 5.8 Example of *kl*-cut input in the remapping approach.

Table 5.2 Original and factorized logic expressions of the *kl*-cut remapping example.

Method	Logic expressions	Mapped circuit area	Difference
Original	o0 = (!i0 + !i3) o1 = ((!i3 * ((!i0 * i1 * i4) + (!i1 * !i4 * i0))) + (i2 * i3)) o2 = (!i3 * ((!i1 * !i4) + (i1 * i4))) o3 = ((!i1 * !i4) + (i1 * i4))	9.792 um²	0%
Multiple output	o0=(!i0 + !i3) o1=((i2 * i3) + (**o2** * ((!i0 * i1) + (i0 * !i1)))) o2=(!i3 * **o3**) o3=!((i1 * !i4) + (!i1 * i4))	6.732 um²	-31.25%
Multiple output with *xor*	o0=(!i0 + !i3) o1=((i2 * i3) + ((!i0 + !**o2**) ^ (!i1 + !**o2**))) o2=(!i3 * **o3**) o3=!(i1 ^ i4)}	9.18 um²	-6.25%
Single output	o0=(!i0 + !i3) o1=((i2 * i3) + (!i3 * ((i0 * (!i1 * !i4)) + (!i0 * (i1 * i4))))) o2=(!i3 * !((i1 * !i4) + (!i1 * i4))) o3=!((i1 * !i4) + (!i1 * i4))	9.792 um²	0%
Single output with *xor*	o0=(!i0 + !i3) o1=((i2 * i3) + (!i3 * ((i0 * (!i1 * !i4)) + (!i0 * (i1 * i4))))) o2=(!i3 * !(i1 ^ i4)) o3=!(i1 ^ i4)}	10.404 um²	+6.25%

5.2.7 KL-cut replacement

After remapping all *kl*-cuts, the cuts are ordered from the largest to the lowest gain, and a greedy selection is performed. It is necessary to save a copy of the current circuit, in the case of the resulting circuit does not respect the timing constraints, so the cut replacement can be undone. A parenthesis for the implementation of this copy: in Java, it must be done with a copy constructor instead of a simple clone method; otherwise, there is a memory leak problem which causes the program to fail or increases a lot the runtime.

During the cut replacement, if the cuts are grouped in P-classes, it is necessary to use the signature information of the inputs and outputs to correct the order. This information is fundamental to keep the correct logic equivalence. Then, the instances and the intermediate wires contained in the original *kl*-cut are removed, the new intermediate wires and instances are added, and the inputs and outputs of the *kl*-cut are connected to the circuit.

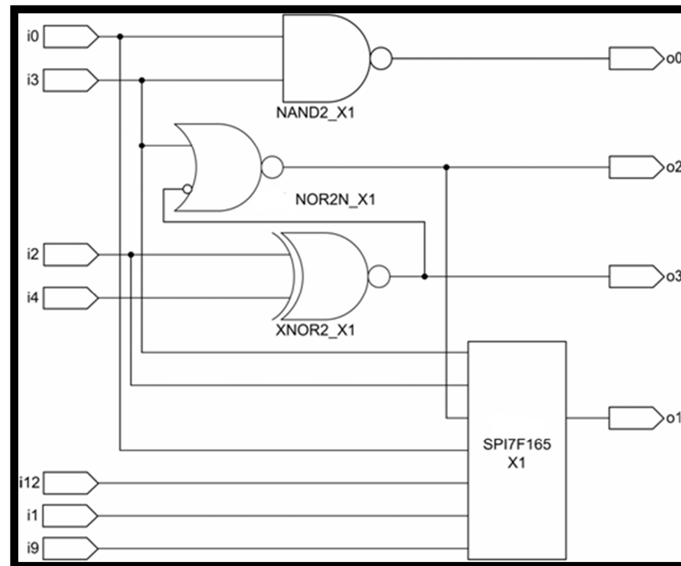


Figure 5.9 Example of Figure 5.8 remapped by the proposed approach.

5.2.7.1 KL-cuts overlapping

Two *kl*-cuts do not overlap if: (1) their set of logic gates covered is disjoint; and (2) their intermediate wires and input/output sets are disjoint, since an intermediate wire of a *kl*-cut cannot be used by another after restructuring its internal logic. The overwhelming majority of the cases are due to (1), the condition (2) is necessary due to the *polarity don't cares*.

The overlapping of *kl*-cuts happens due to the *kl*-cuts algorithm characteristic of finding different supports for the same node. This *kl*-cut characteristic finds optimizations that are not found by other techniques, since several different sub-circuits are found in the same part of the circuit, but it is necessary to check the overlapping, since two overlapped cuts cannot be replaced at the same time.

5.2.7.2 STA checking engine

A *kl*-cut remapped can improve a lot the cost function, but it is necessary to guarantee that the design still meets the performance requirements after replacing the cut back in the netlist. In this work, the *kl*-cuts are remapped disconsidering timing, and timing is checked only when the cut is replaced. The replacement is undone if the delay increased (higher than the design constraints). However, even if the *kl*-cuts were remapped considering timing, this check after replacing a cut would still be necessary, since a cut replacement may affect negatively the timing in other region of the circuit.

There are four different paths that must be checked, as seen in Table 5.3. All endpoints are analysed and the worst delay of all paths is returned. The circuit is checked against the clock period defined in the SDC file. A command to report timing

was created, as seen in Figure 5.10, in order to help the user to debug the critical path delay.

5.2.7.3 Greedy choice and iterative remap

During the development of this work, it was tried to develop a better choice algorithm. However, the high computational effort did not correspond in great gains in the cost function desired. The selection of the best subset of *kl*-cuts that do not overlap can be reduced to a graph coloring problem, which is NP-complete. Moreover, when a *kl*-cut is replaced, the timing of all the following *kl*-cut replacements is affected. Therefore, the selection of the best subset of *kl*-cuts that (1) do not overlap and (2) keep the timing constraints attained is very complex and not worth computationally.

The option to perform a greedy choice and more remapping iterations resulted in better runtime and results, if compared to a better selection algorithm. Since the objective of the remapping flow is to improve a cost function, the proposed iterative algorithm with a greedy choice is able to achieve that, in all cases, with a reasonable runtime.

Table 5.3 Different paths analysed in the STA check.

Path type	**Worst delay**
input-to-register	input delay+combinational circuit delay+clock uncertainty+register setup
input-to-output	input delay+combinational circuit delay+clock uncertainty+output delay
register-to-register	register delay+combinational circuit delay+clock uncertainty+ register setup
register-to-output	register delay+combinational circuit delay+clock uncertainty+output delay

5.3 Conclusion

This chapter presented an iterative remapping flow, based on local transformations using *kl*-cuts. The proposed approach was implemented in an operational tool called KLever2, and it is able to reduce a cost function such as area, while respecting timing constraints. A complete suite of implementations and knownledge was necessary to implement such a tool: a Liberty parser and library data structure; a structural Verilog parser and mapped circuit data structure; an SDC parser and data structure; *k*-cut and *kl*-cut enumeration algorithms, parser and data structure; an extension of a multiple output P-signature algorithm, in order to consider *polarity don't cares*; an STA engine, with results comparable to a commercial tool; Boolean factoring aggressive algorithms; and logic tree mapping algorithms. The proposed flow is composed of many heuristics. The quality of results is due to a combination of the following attributes: (1) use of *kl*-cuts which minimize the support of the Boolean functions; (2) extraction of full context, by using *kl*-cuts instead of *k*-cuts; (3) use of aggressive Boolean optimization techniques to optimize sub-circuits (*kl*-cuts); and (4) allow only substitutions that do not impact negatively on the timing constraints.

```
C:\Windows\system32\cmd.exe

C:\test>java -jar klever2.jar -f sta.script
Klever2 - KL-cuts for Verilog
>read_liberty -f base_library.lib -o lib
>read_verilog -v c432.netlist.v -l lib -o design
>read_sdc -f c432.sdc -d design -o sdc
>
>help report_timing
usage: report_timing -d <design> [-n <number>] [-s <sdc>]
Report worst case delay for a design.
 -d <design>    Design description
 -n <number>    Number of paths
 -s <sdc>       SDC name
>report_timing -d design -s sdc -n 2
Report delay of circuit c432 for 2 path(s):

Point                     Wire    Incr.       Delay       Type
------------------------------------------------------------------
input/input (input)       N108    0.000000    0.000000    fall
U117/Q (INV_X1)           n30     0.012739    0.012739    fall
U173/Q (NOR2_X1)          n49     0.023326    0.043810    fall
U163/Q (NOR4_X1)          n97     0.072803    0.112953    rise
U105/Q (AND4_X1)          n103    0.105255    0.218208    rise
U154/Q (CLKINV_X1)        N223    0.075691    0.293899    fall
U124/Q (NAND2_X0D5)       n72     0.055027    0.348926    rise
U176/Q (NAND3_X1)         n77     0.038805    0.387731    fall
U115/Q (NAND4_X1)         N329    0.139336    0.476683    fall
U109/Q (INV_X2)           n7      0.075844    0.552526    rise
U149/Q (OAI21_X0D5)       n87     0.043250    0.595777    fall
U38/Q (OA33_X1)           n58     0.089164    0.684941    fall
U169/Q (NAND4_X1)         N370    0.085537    0.731845    fall
U116/Q (INV_X1)           n6      0.062442    0.794286    rise
U127/Q (OAI221_X0D5)      n39     0.073612    0.867899    fall
U158/Q (INV_X1)           n4      0.039879    0.907778    rise
U168/Q (OAI221_X1)        n31     0.034631    0.942409    fall
U1/Q (AO21_X1)            N432    0.048331    0.990739    fall
output/output (output)    N432    0.000000    0.990739    fall

Point                     Wire    Incr.       Delay       Type
------------------------------------------------------------------
input/input (input)       N108    0.000000    0.000000    fall
U117/Q (INV_X1)           n30     0.012739    0.012739    fall
U173/Q (NOR2_X1)          n49     0.023326    0.043810    fall
U163/Q (NOR4_X1)          n97     0.072803    0.112953    rise
U105/Q (AND4_X1)          n103    0.105255    0.218208    rise
U154/Q (CLKINV_X1)        N223    0.075691    0.293899    fall
U124/Q (NAND2_X0D5)       n72     0.055027    0.348926    rise
U176/Q (NAND3_X1)         n77     0.038805    0.387731    fall
U115/Q (NAND4_X1)         N329    0.139336    0.476683    fall
U109/Q (INV_X2)           n7      0.075844    0.552526    rise
U149/Q (OAI21_X0D5)       n87     0.043250    0.595777    fall
U38/Q (OA33_X1)           n58     0.089164    0.684941    fall
U169/Q (NAND4_X1)         N370    0.085537    0.731845    fall
U116/Q (INV_X1)           n6      0.062442    0.794286    rise
U127/Q (OAI221_X0D5)      n39     0.073612    0.867899    fall
U158/Q (INV_X1)           n4      0.039879    0.907778    rise
U160/Q (NOR4_X1)          n40     0.066693    0.951097    fall
U197/Q (AOI31_X1)         N421    0.038893    0.989989    rise
output/output (output)    N421    0.000000    0.989989    rise

>
>quit
```

Figure 5.10 Report timing performed in the c432 benchmark.

6. MANUFACTURABILITY COST FUNCTION

This chapter presents the third contribution of this work, which is a cost function to use during logic synthesis. The cost function tries to improve the number of good chips per wafer during logic synthesis, considering lithography printability and other sources of yield loss.

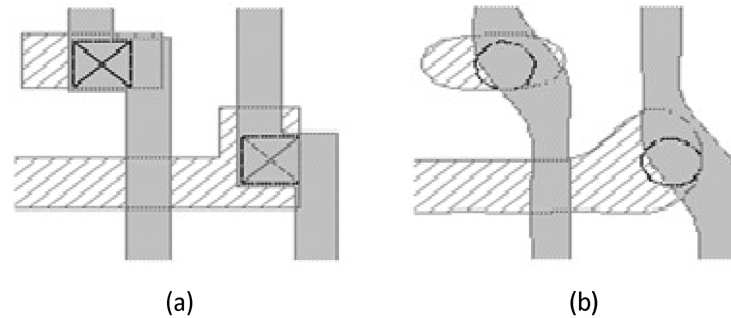
6.1 Design for manufacturability

Integrated circuit (IC) design tools target a well-defined circuit layout, which take into account different environment conditions. Nevertheless, due to the presence of defects and variations during the IC fabrication, the final circuit may be very different from the circuit provided by the CAD (computer-aided design) tools. Additionally, the well-defined circuit layout may not work after the fabrication process. In order to increase the number of good dies per wafer, several techniques have been created, leading to the emergence of the *Design for Manufacturability* (DFM) field (AITKEN, 2006). DFM methods include estimation of yield prior to fabrication, enabling to solve manufacturability problems earlier in the design flow.

One of the sources of defects and variations in IC manufacturing is the lithography process resolution. The CMOS technology scaling down is forcing the lithography to transfer sub-wavelength design features. The current lithography systems, which have a wavelength of 193nm, are being operated in 20nm technologies (WONG, 2009). The result is that the printability of layout shapes is very limited as seen in Figure 6.1. Figure 6.1a shows the square features expected in the designed layout, and its deformed lithography result is in Figure 6.1b, showing rounded shapes and potential problems in the contacts. Several resolution enhancement techniques (RETs) are used to improve the layout printability, such as optical proximity corrections (OPC) and phase shift mask (PSM). However, the post-layout processing steps are not able to exploit all the benefits of such techniques (KHETERPAL, 2005). Moreover, the cost of RETs is also very high if applied to a non-regular layout. Consequently, the use of regular layouts can increase a lot the effectiveness of RETs, while keeping them at a reasonable cost. Nevertheless, notice that there is a significant area overhead by introducing regular layouts, which also affects the number of good dies per wafer.

It is known that DFM techniques applied in post-layout phase improve yield in up to 10%, according to (NARDI, 2004). But, a general rule is that a higher level of abstraction implies higher possibilities of improvement of a cost function. Following this general rule, several works explored improvements in yield earlier in the design flow (HEINEKEN, 1998; SHAIK, 2000; NARDI, 2004). In (HEINEKEN, 1998), in-place substitution of the original cells by yield-optimized cells was proposed, preserving the gate-level netlist, and therefore obtaining limited results. The necessity of

considering manufacturability during logic synthesis has already been stated (SHAIK, 2000), and implemented. The pioneer work of Nardi (2004) considered yield as cost function in a logic synthesis tool and generating an yield optimized gate-level netlist. However, regularity and lithography printability was not considered, and thus the results may not be so relevant for sub wavelength technologies.



(a) (b)

Figure 6.1 Comparison between (a) designed layout and (b) lithography simulation of the designed layout.

No previous work considers the tradeoff of cells with different levels of regularity and area overheads, in order to improve overall design yield during logic synthesis. This work presents a novel yield model for ICs, which considers lithography printability problems (WUU, 2009; GÓMEZ, 2010; DING, 2011; SUNDARESWARAN, 2011) as a source of yield loss. Moreover, a technology remapping approach considering this yield model as cost function is proposed and implemented, using the *kl*-cuts methodology. The methodology proposed by this work can take advantage of regularity for different degrees of severity of lithography hot spots, in order to improve the number of good dies per wafer.

6.2 Yield background

IC *yield* is the amount of the ICs that meet all design specifications divided by the total number of manufactured ICs. When the IC works, but does not meet all performance specifications, this is known as a *parametric yield loss*. The *catastrophic yield loss* refers to problems that cause the product to fail completely (WONG, 2009). Parametric yield loss is generally solved through statistical methods, which try to explore the intrinsic statistical characteristics of fabrication process variations (SINGH, 2005). The catastrophic yield loss has several sources, such as the class of the manufacturing clean room and lithography issues. This work only considers catastrophic yield, and from this point on, the word catastrophic will be suppressed.

The manufacturing clean room dust particles distribution and size is one of the major sources of yield loss, especially in current technologies, since a very small particle can cause a full die to fail. The calculation of this source of yield loss has to take into account the probability of a dust particle size to be anywhere in the wafer, considering the statistics of the clean room, and the critical area. Moreover, the critical area also has to be defined statistically, as it is defined by processes such as chemical metal polishing (CMP) (LUO, 2006), etching and lithography (BUBEL, 1995).

The fabrication process variations are other major sources of yield loss. Processes such as CMP can make metal lines more susceptible to fail than they were designed. Over (under) etching can make some features larger (smaller) than expected in the designed layout, leading to potential shorts or breaks. Lithography problems can be

divided into: (1) defects in the masks or lithography process which generate positive (e.g. bridges) or negative (e.g. holes) pattern transfers, and (2) problems due to resolution of sub wavelength pattern transfers (WONG, 2009).

The problems due to dust particles, process variations and lithography defects can be investigated after fabrication and solved, leading to a learning curve in the fabrication processes. That is one of the main obstacles of performing DFM techniques earlier in the design flow, since a fabrication process that used to be the major source of defects during design phase, can be already solved in the manufacturing phase (AITKEN, 2006; WONG, 2009). Nonetheless, as long the sub wavelength lithography exists, the difficulties of resolution will exist, and the quality of resolution will depend on the RETs and the regularity of layout. This work tries to explore this source of yield loss, while taking into account the other sources, in order to improve the number of good dies per wafer.

6.3 Yield model

This section describes the novel yield model proposed by this work. The goal of this yield model is to consider both sources of yield loss: density of defects (process variations, dust particles and lithography defects) and lithography printability. The effort is not on a severe statistical model for yield prediction, but an intuitive cost function to be explored during logic synthesis phase.

The profitability of a process is affected by the number of dies per wafer (*#DW*). The number of dies per wafer depends on the area of the wafer (A_{wafer}) and on the area of the die (A_{die}), which is given by Equation (6.1).

$$\#DW \cong \frac{A_{wafer}}{A_{die}} \tag{6.1}$$

The profitability of a process is given by the number of good dies per wafer (*#GDW*). The *#GDW* is the number of dies per wafer (*#DW*) multiplied by the yield, as expressed by Equation (6.2).

$$\#GDW = yield \cdot \#DW \tag{6.2}$$

The yield due the presence of defects during the manufacturing processes depends on the critical area (*ca*) and the defect density (*dd*). The critical area (*ca*) of a circuit can be defined as the sum of the critical area of all circuit cells, as defined in Equation (6.3).

$$ca = \sum_{i=1}^{n} (CAci) \tag{6.3}$$

The critical area of a circuit cell (*CAci*) is a region of the cell that may be affected by defects, leading a circuit to failure. In order to calculate *CAci*, it is necessary to take into account the manufacturing processes that affect the real size of critical area after fabrication (BUBEL, 1995; LUO, 2006). Moreover, the *CAci* value must consider the types and diameters of the defects, i.e. the defects that can effectively lead a circuit to fail.

The defect density (*dd*) is inherently associated with the process, and it depends on factors such as the class of the clean room and manufacturing processes variations. The defect density value must consider not only statistical results of the foundry, but also a probability function: a defect can appear in any part of the wafer.

Yield due to density of defects (*dd_yield*) can be defined as a Poisson distribution of the *ca* and *dd* product, as shown in Equation (6.4). Notice that this model is pessimistic for yield value prediction, mainly because defects are not uniformly distributed across the wafer but tend to cluster. There are numerous models that predict yield more accurately (KOREN, 1998), but the objective of this work does not lay on the accuracy of the model. The goal is to know how logic synthesis can be used in order to improve manufacturability.

$$dd_yield = e^{-(dd \cdot ca)} \tag{6.4}$$

It is important to notice that this formulation does not consider the dependency on the lithography printability of the cells. A large number of authors state that having more regular layouts increase yield due to improve in lithography printability (KHETERPAL, 2005), but this behavior is not well captured in Equation (6.1) to Equation (6.4). By using this model, since regular cells have a larger area, regular layouts would reduce the predicted *#GDW* due to area overhead. Under these considerations, the previously described yield could be stated as the yield related to the defect density. In the following, an additional factor to the formulation is proposed, taking into account the criticality of lithography hot spots (*chs*) of the circuit layout. The factor *chs* depends solely on the circuit layout. On the opposite, the severity of lithography hot spots depends on the fabrication technology, the RETs used, and the quality (and calibration) of the lithography system. Consequently, a severity of lithography resolution defects (*sld*) factor is introduced in the formulation. This analysis results in a different yield formulation, called herein as *lhs_yield*, which is related to lithography printability and it is given by Equation (6.5).

$$lhs_yield = e^{-(sld \cdot chs)} \tag{6.5}$$

Notice that Equation (6.5), meant for lithography, is very similar to Equation (6.4), meant for defect density. The criticality of lithography hot spots (*chs*) used in Equation (6.6) can be expressed as the sum of the cell criticality of lithography hot spots (*CHSci*) for all cells instantiated in the circuit, as describe in Equation (6.7). In order to define *CHSci*, it is necessary to evaluate the lithography hot spots. Lithography hot spots are patterns in the layout which are more susceptible to suffer a large variation during lithography (WUU, 2009; GÓMEZ, 2010; DING, 2011; SUNDARESWARAN, 2011).

$$chs = \sum_{i=1}^{n} (CHSci) \tag{6.6}$$

Equation (6.4) and Equation (6.5) give two different formulations for yield, from different yield loss causes. These formulas can be combined into a new yield formulation shown in Equation (6.7), called herein as total yield (*ty*). The substitution of Equation (6.3) to Equation (6.6) into Equation (6.7) gives the complete formulation for total yield, illustrated in Equation (6.8). Notice that in Equation (6.8) the total yield (*ty*) depends on

two foundry technology parameters: the defect density and the severity of lithography resolution defects.

$$ty = dd_yield \cdot lhs_yield \tag{6.7}$$

$$ty = e^{-(dd \cdot \sum_{i=1}^{n}(CAci) + sld \cdot \sum_{i=1}^{n}(CHSci))} \tag{6.8}$$

The total yield in Equation (6.8) can be used to compose a cost function to take both yield loss sources into account while performing technology mapping. This can be done by substituting Equation (6.1) and Equation (6.8) into Equation (6.2). The resulting cost function expresses the number of good dies per wafer (*#GDW*) and it is illustrated in Equation (6.9).

$$\#GDW = \frac{A_{wafer}}{A_{die}} \cdot e^{-(dd \cdot \sum_{i=1}^{n}(CAci) + sld \cdot \sum_{i=1}^{n}(CHSci))} \tag{6.9}$$

The *#GDW* has to be maximized during technology mapping, in order to improve manufacturability. The formulation in Equation (6.9) allows to tradeoff lithography effects, defect density and total area while maximizing the number of good dies per wafer.

6.4 Yield as a cost function

The yield model has been proposed, it is important to discuss about the values to be used in a cost function. This section is intended to explain the rationale of the choice of numbers for *dd* and *sld*; as well as the range of values for *CHSci* for the cells. The defect density (*dd*) is a function of the expected number of critical area defects *(#cad*) expected in a wafer, given in Equation (6.10). The *#cad* must be calculated through a statistical and probabilistic analysis.

$$dd = \frac{\#cad}{A_{wafer}} \tag{6.10}$$

The *sld* has to emulate a similar behavior to *dd* for the *lhs_yield*. The purpose of the *sld* parameter in the proposed formulation is to increase or decrease the number of expected lithography resolution defects (*#LD*) for a given technology, compared to the expected number of critical area defects (*#cad*). For *sld*=1, it is assumed that *#LD* is similar to *#cad*, as demonstrated in Equation (6.11). As a consequence, the *sld* can be modeled as the ratio between the number of expected lithography resolution defects (*#LD*) and the number of expected critical area defects (*#cad*), as expressed by Equation (6.12).

$$sld = 1 \rightarrow dld = \frac{\#LD}{A_{wafer}} \cong dd = \frac{\#cad}{A_{wafer}} \tag{6.11}$$

$$sld \cong \frac{\#LD}{\#cad} \rightarrow \#LD(sld = 1) \cong \#cad \tag{6.12}$$

The purpose of the criticality of hot spots (*CHSci*) parameter is to express how the quality of cell layouts influences *#LD*. The parameter *CHSci* is derived from the number of lithography resolution defects expected in a wafer completely filled with instances of a cell *ci* (*#LDci*). The parameter *#LDci* is called number of lithography resolution defects induced by a cell. The relationship between *CHSci* and *#LDci* is expressed in Equation (6.13). A cell layout with good printability has $\#LDci < \#LD$, a cell with bad printability has $\#LDci > \#LD$, and a cell with average printability has $\#LDci \cong \#LD$.

$$CHSci \cdot \frac{A_{wafer}}{A_{cell}} = \#LDci \rightarrow CHSci \cong \#LDci \cdot \frac{A_{cell}}{A_{wafer}} \tag{6.13}$$

Consider as an example, a wafer with 600 cm^2 of area, which can produce approximately 150 dies of 4cm^2, and the *#cad* equals to 15. This scenario implies a defect density of 15 defects/wafer, or 0.025 defects/cm^2. Consider also that the technology has a *sld* equals to 1, meaning that additional 15 defects will happen in the wafer due to lithography resolution problems, i.e. *#LD* equals to 15 defects/wafer on average. Additionally, assume that the *#LDci* of cells can vary around *#LD*, depending on the printability of the cell layout.

Table 6.1 shows values of *CHSci* computed for cell versions with different degrees of printability. The cells considered as reference layout do not include any lithography consideration. Some cells are restricted to be designed on a regularly spaced grid, but can use two-dimensional features (2D-grid). Litho-friendly cells are restricted to use one-dimensional features (1D-restr). The *CHSci* values in Table 6.1 can be used to compute the yield given by the different choices of layout.

Table 6.1 Values of *CHSci* derived according to Equation (6.13) considering a wafer of 600 cm².

Cell Function	**Layout type**	**Area (μm^2)**	**Printability**	**#LDci**	**CHSci**
and2	1D-restr	1.57	good	7	1.83e-10
and2	2D-grid	1.41	average	15	3.53e-10
and2	reference	1.22	bad	30	6.10e-10
inv	1D-restr	0.784	good	7	0.92e-11
inv	2D-grid	0. 706	average	15	1.77e-10
inv	reference	0.612	bad	30	3.06e-10

Assume that a designer wants to verify the yield of dies that would have 4cm^2 with reference layouts. This results in a number of instances of the reference layout given by Equation (6.14).

$$\#inst = \frac{A_{die}}{A_{reference}} \tag{6.14}$$

The yield values for 1D-restr and 2D-grid layouts are computed for the same number of instances. The computation of the yield of several instances of the same benchmark tied together can be computed as described in Equation (6.15).

$$ty(\#inst) = ty(1)^{\#inst} \tag{6.15}$$

Results scaled for reference layouts with 4cm^2 are shown in Table 6.2. Notice that the option that produces the larger *#GDW* is the reference cells for this scenario (*sld*=1) due

to a larger *#DW*, even with the worst yield. The results in Table 6.2 are affected by the *sld* value. A *sld* equals to 2 would make 1D-restr similar to reference cells, in terms of *#GDW*. A *sld* equals to 3 would make 1D-restr cells better than the other layout types cells, in terms of *#GDW,* justifying the use of regular layout cells.

Table 6.2. Values of *#GDW* considering a die of 4 cm² on a wafer of 600 cm² for the reference cells, and the same number of cell instances for 1D-restr and 2D-restr cells; *sld*=1.

Cell Function	Layout type	*#inst*	Area (cm^2)	*#DW*	Yield	*#GDW*
and2	1D-restr	3.28e8	5.15	116	82.80%	96
and2	2D-grid	3.28e8	4.62	129	79.36%	102
and2	reference	3.28e8	4.00	150	74.08%	111
inv	1D-restr	6.54e8	5.12	117	82.87%	96
inv	2D-grid	6.54e8	4.61	130	79.40%	103
inv	reference	6.54e8	4.00	150	74.08%	111

6.5 Yield remapping tool

The information of lithography criticality of hot spots and critical area is passed to the tool when the library is read, associating the values of the cells as they are read from the Liberty files. The information about wafer size, density of defects and severity of lithography defects is passed through *set_technology_info* command, as seen in Figure 6.2.

The standard yield-aware design flow is shown in Figure 6.3a. The logic synthesis (targeting area, performance and power) generates a gate-level netlist, which goes to placement and routing to create the design layout. The only concern on yield improvements happens on the post-layout phase, where in-place yield enhancements are performed. The flow proposed by Nardi (2004) is shown on Figure 6.3b, where the manufacturability is considered during logic synthesis, performing a more global optimization of yield. However, it is not a good approach to simply replace the commercial tool, or its cost function. For instance, the area results can be much worse and the number of dies per wafer (*#DW*) may decrease a lot. Furthermore, the lithography resolution problems are simply ignored in (NARDI, 2004).

This work proposes a technology remapping approach after the usual logic synthesis process performed by a commercial tool. The proposed tool improves a cost function of interest, as seen in Figure 6.3c. In this case, the cost function used is the number of good dies per wafer (*#GDW*), according to Equation (6.9). Besides the methodology itself, the proposed approach takes into account the tradeoff between lithography printability and area overhead. Additionally, the proposed method can be applied in design sub modules that are statistically more susceptible to yield loss, for instance.

```
C:\Windows\system32\cmd.exe

C:\test>java -jar klever2.jar
Klever2 - KL-cuts for Verilog
>help set_technology_info
usage: set_technology_info -d <density_of_defects> -s
       <severity_of_lithography_defects> -w <wafer_size>
Set density of defects
 -d <density_of_defects>                   Density of defects (1/cm2)
 -s <severity_of_lithography_defects>      Severity of lithography defects
 -w <wafer_size>                           Useful area of wafer (cm2)
>_
```

Figure 6.2 Passing technology information to the remapping tool.

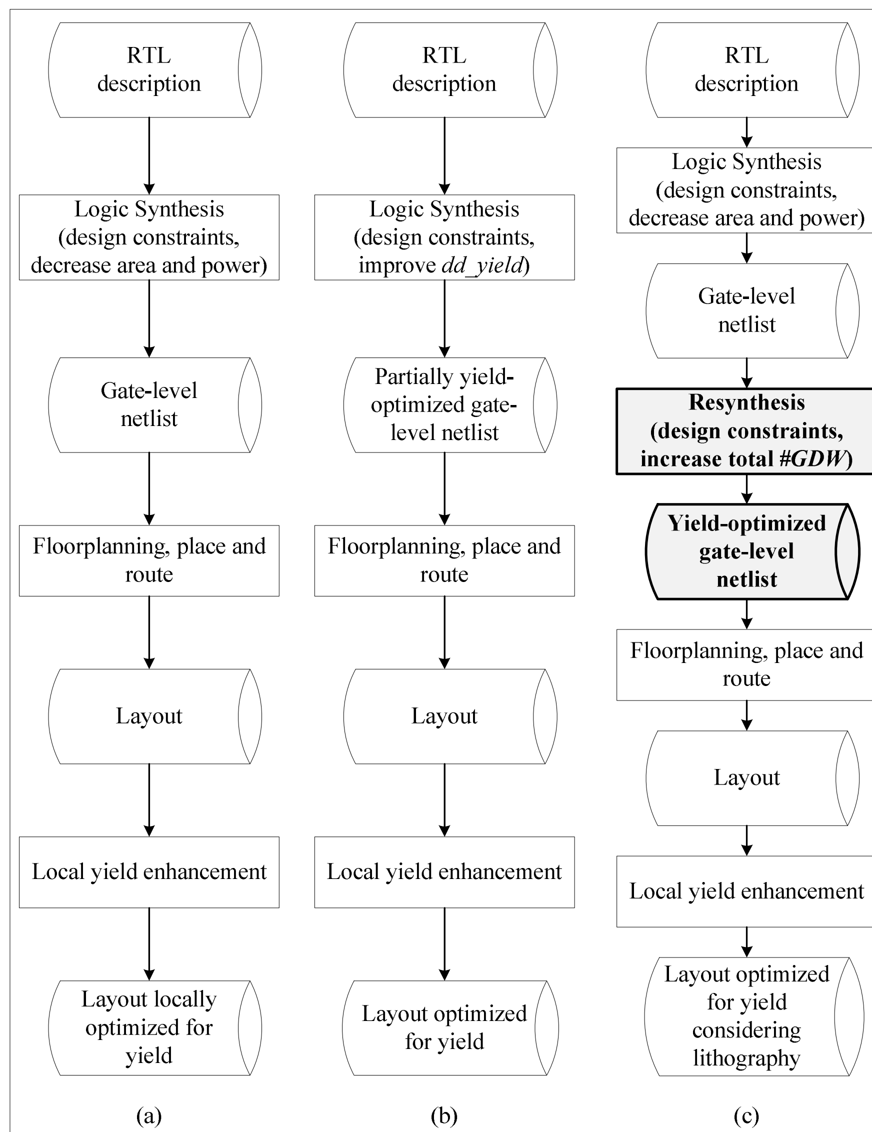


Figure 6.3 (a) Standard “yield-aware” flow, (b) Flow proposed by Nardi (2004), and (c) the flow propose by this work.

6.6 Conclusion

This chapter presented the introduction of a novel yield model for integrated circuits manufacturing, considering lithography printability and density of defects, which can be used as a cost function for logic synthesis process. The proposed methodology establishes a new standpoint in the field of regular layout by introducing a metric to tradeoff area and printability of layouts. Many of the previous works completely ignored these tradeoffs and simply pointed out that regularity is expected to improve yield somehow, without presenting or discussing metrics. Therefore, a great contribution of this work is to propose a discussion about the tradeoffs between different area overheads and different levels of lithography printability for regular layouts.

7. RESULTS

This chapter presents the results of the experiments of this work. Section 7.1 presents the results of the STA engine developed in order to guarantee that the design will still work at the expected performance after the remapping approach. Section 7.2 shows experiments trying to improve area results of commercial tools. Section 0 presents the results of experiments improving the cost function proposed in Chapter 6.

7.1 Precision of STA engine

Sequential digital circuits must be analyzed in order to check if there are no timing violations. This analysis is very important to determine if the design works correctly at the expected performance. The static timing analysis is a method to analyze timing, considering only the worst case at each logic gate of the design (BHASKER, 2009). Since CMOS logic gates have different rise and fall characteristics, both cases must be considered.

The STA tool developed for this work is implemented using the nonlinear delay model (NLDM). NLDM determines the output transition time and the output delay of a logic gate through the input transition time of the gate inputs and the capacitance load at the gate output (BHASKER, 2009). The cells timing information is read from Liberty files of the cell library, and the timing constraints are read from a Synopsys Design Constraints (SDC) file (BHATNAGAR, 2001).

In order to identify the critical path, the delay is propagated from the inputs (or registers outputs) to the outputs (or registers inputs), considering only the worst case timing arc at each cell (for rise and fall). The worst delay of a design is the largest delay from all endpoints, i.e. circuit outputs and register inputs.

With the purpose of validating the STA engine, several benchmarks and SDC files have been checked. And for the benchmarks analyzed, no difference in results was observed between the STA tool developed and commercial STA tools. Table 7.1 shows the differences on worst delay analysis between a commercial STA tool and the proposed STA engine, which is basically due to number rounding.

7.2 Area as a cost function

For evaluation of the proposed methodology, the first experiments were on reducing the area. Area is a very good cost function, since represents a direct cost in the final circuit, and also it has a very good correlation: decreasing the area of sub-parts will certainly decreases the whole. The following experiments were performed on a computer with a Core i5 processor, and 4GB of RAM. The commercial logic synthesis tools results were obtained with versions of 2006.

Table 7.1 Comparison of the worst delay given by a commercial STA tool and the STA engine developed for this work.

Benchmark	Commercial STA tool delay (ns)	This paper STA delay (ns)	Diff (ps)
c1355	0.9929	0.9929	0.011
c1908	0.9867	0.9867	0.024
c1908a	0.9725	0.9725	0.035
c2670	0.9806	0.9806	-0.010
c2670a	0.9409	0.9409	0.054
c3540	1.0001	1.0001	0.031
c3540a	0.9978	0.9978	0.090
c432	0.9908	0.9908	-0.069
c499	0.7742	0.7743	0.059
c5315	1.0007	1.0008	0.118
c5315a	0.988	0.988	0.053
c6288	1.9997	2.0007	1.018
c7552	0.9775	0.9776	0.098
c880a	0.8019	0.8023	0.365

7.2.1 Libraries used for area experiments

Two standard cell libraries were used in the experiments: a base library and an extended library. The technology node is 40nm, and the cell libraries were generated by a commercial automatic library generation tool. The base library contains 266 logic gates and a total of 49 different logic functions of combinational cells, which is a representative of common commercial cell libraries.

The extended library is composed of the base library cells plus all cells with up to three transistors in series and up to three transistors in parallel. In the extended library, there are 132 more logic functions than the base library and a total of 181 different logic functions of combinational cells. For this work, we will only consider minimal drive strength cells (X1), for simplification.

The use of an extended library in the experiments is to measure the use of complex logic gates with commercial tools, comparing to the proposed methodology. The use of complex gates can potentially reduce circuit area, but they have to be chosen wisely to preserve timing constraints while remapping.

Table 7.2 Libraries used for area reduction experiments.

Nickname	# of cells	# of functions	Description
Base library	266	49	Common cells found on commercial cell libraries
SP33	174	174	All possible cells using at most 3 transistors in series/parallel
Extended library	398	181	Base library plus SP33

7.2.2 ISCAS benchmarks area results

ISCAS benchmarks are simple circuits well-known in logic synthesis field, and therefore were chosen to evaluate the STA engine at first, and to evaluate the first results of the proposed remapping flow. The ISCAS benchmarks are divided in ISCAS'85 set, which are basically simple combinational circuits, and in ISCAS'89 set, which contain simple and complex sequential circuits. The both sets were used for the

following experiments. The results were obtained with unlimited number of remapping iterations and *k*=5.

7.2.2.1 ISCAS'85 benchmarks area results

The delay constraint and the results of the synthesis using commercial tool A for ISCAS'85 benchmarks (IWLS 2005 benchmarks, 2012) are in Table 7.3. Notice that two syntheses were performed for each benchmark: a synthesis using base library (Set 1) and a synthesis using extended library (Set 2). The delay constraint was defined based on the mapping of the circuits without timing constraints.

It is important to see that commercial tool A is not always able to use the extra cells in order to improve area. On the contrary, commercial tool A increases area for several benchmarks when extra cells are available, up to 20%. For the following results, the Set 1 results were used as reference.

Table 7.3 ISCAS'85 benchmarks synthesized with commercial tool A.

Benchmark	Delay Constraint (ns)	Base library (Set 1)		Extended library (Set 2)	
		Area (μm²)	Delay (ns)	Area Diff	Delay (ns)
c1355	1.0	289.4760	0.9929	-5.71%	0.9639
c1908	1.0	277.2360	0.9867	+2.10%	0.9994
c1908a	1.0	200.7360	0.9725	+0.15%	0.9951
c2670	1.0	372.0960	0.9806	-0.99%	0.9957
c2670a	1.0	386.7840	0.9409	-1.35%	0.9964
c3540	1.0	660.6540	1.0001	+9.91%	1.0005
c3540a	1.0	559.9800	0.9978	+19.78%	0.9999
c432	1.0	121.4820	0.9907	+3.78%	0.9991
c499	1.0	261.6300	0.7743	-0.58%	0.7432
c5315	1.0	921.9780	1.0008	-2.62%	0.9986
c5315a	1.0	896.2740	0.9880	-3.35%	0.9988
c6288	2.0	2572.2583	1.9996	+18.81%	2.0007
c7552	1.0	1163.7180	0.9776	+1.24%	1.0000
c880a	1.0	229.5000	0.8023	-3.33%	0.9219
Average	-	-	-	+2.70%	-
Worst	-	-	-	+19.78%	-
Best	-	-	-	-5.71%	-

In order to check if the results could be further improved with a different commercial logic synthesis tool, the benchmarks were synthesized using commercial tool B and the results are shown in Table 7.4. The area results were much worse than the results obtained with commercial tool A. The proposed methodology was used to remap only the commercial tool A results, since the results are already very good, and therefore harder to improve.

The gate netlists generated by commercial tool A were then remapped using the proposed methodology, and the results shown in

Table 7.5 were generated. Notice that area was reduced in all cases (up to 15%), and almost all cases were using the extra complex gates. The average runtime was about 8 minutes, the best case runtime was 2 seconds and the worst case runtime was 70 minutes (for "c6288" benchmark, which is the biggest and therefore more *kl*-cuts and more iterations). Notice that the area results are being compared with the circuits from

Set 1, therefore the remapping results of Set 2 reduced the area compared to Set 2, but the area results are still larger than Set 1 results in some cases.

Table 7.4 ISCAS'85 benchmarks synthesized with commercial tool B.

Benchmark	Delay Constraint (ns)	Base library		Extended library	
		Area Diff	Delay (ns)	Area Diff	Delay (ns)
c1355	1.0	+42.28%	0.5870	+42.28%	0.5872
c1908	1.0	+44.59%	0.9407	+43.49%	0.9565
c1908a	1.0	+56.25%	0.8206	+56.86%	0.8137
c2670	1.0	+89.97%	0.6351	+90.05%	0.6354
c2670a	1.0	+56.09%	0.6161	+56.01%	0.6233
c3540	1.0	+42.66%	0.9878	+45.58%	0.9890
c3540a	1.0	+35.30%	0.9699	+34.26%	0.9919
c432	1.0	+34.51%	0.9314	+32.75%	0.8893
c499	1.0	+57.43%	0.5871	+57.43%	0.5886
c5315	1.0	+49.05%	0.9883	+50.15%	0.9883
c5315a	1.0	+50.97%	0.9883	+49.37%	0.9828
c6288	2.0	-2.95%	1.9919	-6.38%	1.9919
c7552	1.0	+52.59%	0.9853	+53.14%	0.9854
c880a	1.0	+27.20%	0.6950	+30.13%	0.6748
Average	-	+45.42%	-	+45.37%	-
Worst	-	+89.97%	-	+90.05%	-
Best	-	-2.95%	-	-6.38%	-

Table 7.5 ISCAS'85 benchmarks synthesized with commercial tool A remapped with the proposed methodology.

Benchmark	Remapping of Set 1				Remapping of Set 2	
	Base library		Extended library		Extended library	
	Area Diff	Delay (ns)	Area Diff	Delay (ns)	Area Diff	Delay (ns)
c1355	-6.24%	0.9225	-7.61%	0.9931	-10.99%	0.9309
c1908	-8.06%	0.9564	-8.61%	0.9878	-9.16%	0.9988
c1908a	-3.96%	0.9993	-5.95%	0.9424	-6.86%	0.9496
c2670	-6.50%	0.9938	-7.24%	0.9999	-6.00%	0.9956
c2670a	-7.67%	0.9950	-8.86%	0.9817	-6.88%	0.9998
c3540	-5.60%	0.9997	-7.18%	0.9985	+7.23%	1.0002
c3540a	-4.64%	0.9989	-5.25%	0.9986	+16.78%	1.0000
c432	-5.54%	0.9980	-5.54%	0.9945	-4.03%	0.9967
c499	-2.92%	0.8235	-3.27%	0.8679	-2.92%	0.8942
c5315	-10.16%	0.9985	-11.35%	0.9821	-13.38%	0.9798
c5315a	-11.27%	0.9902	-12.63%	0.9895	-13.52%	0.9670
c6288	-8.69%	2.0000	-8.09%	2.0000	+7.20%	1.9998
c7552	-14.17%	0.9967	-14.99%	0.9999	-13.02%	0.9914
c880a	-6.40%	0.9453	-7.20%	0.9665	-9.07%	0.9878
Average	-7.27%	-	-8.13%	-	-4.62%	-
Worst	-2.92%	-	-3.27%	-	+16.78%	-
Best	-14.17%	-	-14.99%	-	-13.52%	-

7.2.2.2 ISCAS'89 benchmarks area results

The delay constraint and the results of the synthesis using commercial tool A for ISCAS'89 benchmarks (IWLS 2005 benchmarks, 2012) are shown in Table 7.6. Notice that two syntheses were performed for each benchmark: a synthesis using base library (Set 3) and a synthesis using the extended library (Set 4). The delay constraint was

defined based on the mapping of the circuits without timing constraints. Again, commercial tool A was not always able to use the extra cells in order to improve area, increasing area for several benchmarks when extra cells are available. For the following results, the Set 3 results were used as reference.

The gate netlists generated by commercial tool A were then remapped by the proposed methodology and the results shown in Table 7.7 were generated. Notice that combinational area was reduced in all cases (up to 23%), and the majority of cases was using the extra cells. The analysis in sequential benchmarks must consider the combinational area separately from the total area, since the remapping proposed only performs combinational logic restructuring, and therefore may improve only combinational area. The average runtime was about 45 seconds, the best case runtime was 2 seconds, and the worst case runtime was 8 minutes (for "s38584" benchmark). Notice that the area results are being compared with the circuits from Set 3, therefore the remapping results of Set 4 reduced the area compared to Set 4, but the area results are still larger than Set 3 results in some cases.

Table 7.6 ISCAS'89 benchmarks synthesized with commercial tool A.

Benchmark	**Delay Constraint (ns)**	**Commercial tool A**					
		Base library (Set 3)			**Extended library (Set 4)**		
		Comb. Area (μm²)	**Area (μm²)**	**Delay (ns)**	**Comb. Area Diff**	**Area Diff**	**Delay (ns)**
s1196	0.5	384.948	533.6640	0.5002	+5.01%	+3.84%	0.4999
s1238	0.5	378.828	527.5440	0.4995	+2.34%	+1.68%	0.4996
s13207	0.5	700.74	3411.9000	0.5004	-1.44%	-0.31%	0.4993
s1423	2.0	318.24	929.6280	1.9995	+3.94%	+1.48%	1.9973
s1488	0.5	509.184	562.4280	0.5000	-0.96%	-0.98%	0.5001
s1494	0.5	490.824	542.2320	0.4996	+0.81%	+1.07%	0.4996
s15850	0.5	417.69	1522.6560	0.4995	-5.42%	-0.92%	0.5102
s208_1	0.5	38.862	104.9580	0.4968	-6.30%	-2.33%	0.4815
s298	0.5	74.358	190.6380	0.4977	-9.47%	-4.01%	0.4995
s344	0.5	87.21	211.1400	0.4983	+9.82%	+4.35%	0.4996
s349	0.5	85.374	209.3040	0.4993	+16.13%	+7.16%	0.4992
s35932	2.0	5478.624	19755.3600	1.9821	-0.01%	+0.00%	1.9260
s382	0.5	91.494	264.9960	0.4982	+2.01%	+0.69%	0.4999
s38417	1.5	5103.774	18025.5420	1.3684	-2.62%	-0.74%	1.4986
s38584	1.5	4769.622	14403.1140	1.2262	-1.55%	-0.51%	1.4318
s386	0.5	68.544	118.1160	0.4728	-0.89%	-0.52%	0.4974
s400	0.5	93.024	266.5260	0.4993	+4.61%	+1.61%	0.4979
s420_1	0.5	97.92	231.3360	0.4997	+6.25%	+2.65%	0.4982
s444	0.5	87.822	261.3240	0.4985	+8.01%	+2.69%	0.4996
s510	0.5	175.644	226.4400	0.5002	+1.74%	+1.62%	0.4989
s526	0.5	106.488	280.6020	0.4986	+8.91%	+3.16%	0.4993
s526n	0.5	115.974	289.4760	0.4989	-1.32%	-0.53%	0.4993
s5378	0.5	868.428	2223.7020	0.5000	-3.17%	-1.29%	0.5002
s641	0.5	107.712	231.6420	0.4975	+4.55%	+2.11%	0.4995
s713	0.5	105.264	229.1940	0.4978	+8.14%	+3.74%	0.4982
s820	0.5	219.096	262.2420	0.4999	+3.07%	+2.80%	0.4996
s832	0.5	225.828	269.5860	0.4998	-8.54%	-7.60%	0.4985
s838_1	1.5	169.524	433.9080	1.3220	-8.12%	-3.17%	1.4823
s9234_1	1.0	531.828	1729.8180	0.9902	-1.04%	-0.32%	0.9968
Average	-	-	-	-	+1.19%	+0.60%	-
Worst	-	-	-	-	+16.13%	+7.16%	-
Best	-	-	-	-	-9.47%	-7.60%	-

Table 7.7 ISCAS'89 benchmarks synthesized with commercial tool A remapped with the proposed methodology.

Benchmark	Remapping of Set 3						Remapping of Set 4		
	Base library			Extended library			Extended library		
	Comb. Area Diff	**Area Diff**	**Delay (ns)**	**Comb. Area Diff**	**Area Diff**	**Delay (ns)**	**Comb. Area Diff**	**Area Diff**	**Delay (ns)**
s1196	-14.71%	-10.61%	0.4995	-12.16%	-8.77%	0.4999	-10.10%	-7.05%	0.4996
s1238	-11.95%	-8.58%	0.4995	-13.25%	-9.51%	0.4994	-10.82%	-7.77%	0.4992
s13207	-17.25%	-5.72%	0.4994	-17.95%	-5.87%	0.4989	-16.68%	-5.62%	0.5000
s1423	-4.52%	-1.55%	1.9974	-7.31%	-2.50%	1.9998	-4.42%	-1.38%	1.9997
s1488	-6.67%	-6.04%	0.4998	-8.77%	-7.94%	0.4999	-5.59%	-5.17%	0.5001
s1494	-3.68%	-3.33%	0.4997	-3.68%	-3.33%	0.4997	-4.99%	-4.18%	0.4998
s15850	-12.23%	-3.36%	0.4996	-11.28%	-3.09%	0.4999	-21.90%	-5.45%	0.5090
s208_1	-3.15%	-1.17%	0.4983	-8.66%	-3.21%	0.4976	-11.81%	-4.37%	0.4826
s298	-7.41%	-2.89%	0.4987	-9.47%	-3.69%	0.4990	-23.05%	-9.31%	0.4991
s344	-8.77%	-3.62%	0.4973	-8.42%	-3.48%	0.4998	+0.70%	+0.58%	0.4986
s349	-5.38%	-2.19%	0.4974	-5.02%	-2.05%	0.4993	-2.15%	-0.29%	0.4983
s35932	-2.78%	-0.77%	1.9486	-2.93%	-0.81%	1.9480	-2.84%	-0.79%	1.9313
s382	-8.36%	-2.89%	0.4951	-10.03%	-3.46%	0.4950	0.00%	0.00%	0.5000
s38417	-4.04%	-1.14%	1.4958	-4.83%	-1.37%	1.4923	-7.70%	-2.18%	1.4797
s38584	-7.28%	-2.41%	1.4189	-10.27%	-3.40%	1.4971	-10.28%	-3.41%	1.4346
s386	-3.12%	-1.81%	0.4784	-3.12%	-1.81%	0.4784	-4.91%	-2.85%	0.4973
s400	-8.88%	-3.10%	0.4972	-9.21%	-3.21%	0.4984	-4.61%	-1.61%	0.4985
s420_1	-12.81%	-5.42%	0.4969	-14.38%	-6.08%	0.4967	-7.19%	-3.04%	0.5000
s444	-5.92%	-1.99%	0.4978	-4.18%	-1.41%	0.4963	+2.09%	+0.70%	0.4990
s510	-9.41%	-7.30%	0.4999	-15.68%	-12.16%	0.4983	-17.07%	-12.97%	0.4991
s526	-3.45%	-1.31%	0.4967	-4.31%	-1.64%	0.4997	-5.46%	-2.29%	0.4923
s526n	-8.71%	-3.49%	0.4988	-9.50%	-3.81%	0.4970	-8.97%	-3.59%	0.4995
s5378	-9.69%	-3.78%	0.4998	-10.25%	-4.00%	0.4998	-10.50%	-4.16%	0.5002
s641	-22.16%	-10.30%	0.4997	-22.16%	-10.30%	0.4950	-6.25%	-2.91%	0.4995
s713	-17.44%	-8.01%	0.4989	-16.28%	-7.48%	0.4991	-15.70%	-7.21%	0.4979
s820	-11.59%	-9.68%	0.4999	-12.71%	-10.62%	0.4998	-2.09%	-1.52%	0.4998
s832	-12.06%	-10.10%	0.4997	-13.96%	-11.69%	0.4998	-19.65%	-16.91%	0.4994
s838_1	-7.22%	-2.82%	1.4970	-8.30%	-3.24%	1.4909	-13.90%	-5.43%	1.4959
s9234_1	-5.01%	-1.54%	0.9937	-6.04%	-1.86%	0.9305	-6.85%	-2.11%	0.9872
Average	-8.82%	-4.38%	-	-9.80%	-4.89%	-	-8.71%	-4.22%	-
Worst	-2.78%	-0.77%	-	-2.93%	-0.81%	-	+2.09%	+0.70%	-
Best	-22.16%	-10.61%	-	-22.16%	-12.16%	-	-23.05%	-16.91%	-

7.2.3 ITC'99 benchmarks area results

Even though ISCAS benchmarks are still widely used in logic synthesis research, they are very old. In order to give results with more recent benchmarks, ITC 99 benchmarks (IWLS 2005 benchmarks, 2012) were chosen. The delay constraint and the results of the synthesis using commercial tool A for ITC'99 benchmarks are shown in Table 7.8. The delay constraint was defined based on the mapping of the circuits without timing constraints. Notice that two syntheses were performed for each benchmark: a synthesis using the base library (Set 5) and a synthesis using the extended library (Set 6). Commercial tool A was not always able to use the extra cells in order to improve area, and increases area for several benchmarks when extra cells are available, with the worst case of 68% combinational area increase. For the following results, the Set 5 results were used as reference.

The gate netlists generated by commercial tool A were then remapped by the proposed methodology and the results shown in Table 7.9 were generated. Notice that combinational area was reduced in all cases (up to 39%) and the majority of cases was using the extra cells. The average runtime was about 4 hours, the best case runtime was 2 seconds and the worst case runtime was 35 hours (for "b18" benchmark). Notice that the area results are being compared with the circuits from Set 5, therefore the remapping results of Set 6 reduced the area compared to Set 6, but the area results are still larger than Set 5 results in some cases.

Table 7.8 ITC'99 benchmarks synthesized with commercial tool A.

Benchmark	**Delay Constraint (ns)**	**Commercial tool A**					
		Base library (Set 5)			**Extended library (Set 6)**		
		Comb. Area (μm²)	**Area (μm²)**	**Delay (ns)**	**Comb. Area Diff**	**Area Diff**	**Delay (ns)**
b01	0.4	33.660	75.582	0.400	-3.64%	-1.62%	0.396
b02	0.4	14.076	47.124	0.331	-10.87%	-3.25%	0.362
b03	0.8	76.194	324.054	0.793	-0.80%	-0.19%	0.759
b04	1.5	470.322	1015.614	1.403	-2.34%	-1.08%	1.428
b05	1.5	478.584	759.492	1.497	-0.13%	-0.08%	1.498
b06	0.5	33.048	107.406	0.408	-2.78%	-0.85%	0.449
b07	1.0	273.564	637.092	0.923	-1.68%	-0.72%	0.925
b08	0.6	119.340	294.678	0.599	-11.28%	-5.19%	0.598
b09	1.0	83.538	314.874	0.819	-2.93%	-0.78%	0.863
b10	1.0	119.646	260.100	0.949	-5.63%	-2.59%	0.892
b11	0.6	551.412	811.818	0.601	+10.93%	+7.95%	0.601
b12	0.6	833.850	1839.672	0.600	+0.11%	+0.12%	0.601
b13	1.0	179.928	617.814	0.782	-4.59%	-1.34%	0.783
b14	2.0	7794.432	9571.374	2.002	+12.21%	+9.94%	2.004
b14_1	2.0	5443.107	7219.482	2.001	+68.04%	+51.29%	2.004
b15	2.0	5585.112	9031.590	2.011	-0.78%	-0.48%	2.012
b15_1	2.0	5586.948	9034.038	2.011	-0.87%	-0.53%	2.011
b17	2.0	17185.604	28069.199	2.001	-0.31%	-0.17%	2.004
b17_1	2.0	17356.014	28241.964	2.000	-1.17%	-0.73%	2.004
b18	4.0	60159.294	85477.734	4.004	-0.58%	-0.39%	4.001
b18_1	4.0	60645.528	85963.968	4.002	-0.90%	-0.62%	4.002
b19	4.0	123083.910	173741.598	4.002	-0.64%	-0.44%	4.003
b19_1	4.0	123706.711	174415.984	4.002	-0.17%	-0.13%	4.004
b20	2.0	24243.156	27796.428	2.002	-0.86%	-0.74%	2.001
b20_1	2.0	22822.398	26376.894	2.002	+4.55%	+3.97%	2.002
b21	2.0	23019.768	26576.100	2.002	+3.87%	+3.36%	2.002
b21_1	2.0	22498.038	26056.818	2.003	+6.60%	+5.69%	2.003
b22	2.0	37029.366	42364.476	2.001	-2.17%	-1.88%	2.004
b22_1	2.0	25233.508	30567.236	2.003	+13.89%	+11.46%	2.003
Average	-	-	-	-	+2.24%	+2.41%	-
Worst	-	-	-	-	+68.04%	+51.29%	-
Best	-	-	-	-	-11.28%	-5.19%	-

Table 7.9 ITC'99 benchmarks synthesized with commercial tool A remapped with with the proposed methodology.

Benchmark	Remapping of Set 5						Remapping of Set 6		
	Base library			Extended library			Extended library		
	Comb. Area Diff	Area Diff	Delay (ns)	Comb. Area Diff	Area Diff	Delay (ns)	Comb. Area Diff	Area Diff	Delay (ns)
b01	-4.55%	-2.02%	0.3983	-4.55%	-2.02%	0.3986	-9.09%	-4.05%	0.3936
b02	-10.87%	-3.25%	0.3882	-13.04%	-3.90%	0.3575	-19.57%	-5.84%	0.3946
b03	-0.40%	-0.09%	0.7978	-0.40%	-0.09%	0.7978	-2.41%	-0.57%	0.7148
b04	-29.15%	-13.50%	1.4613	-27.46%	-12.71%	1.3935	-27.98%	-12.96%	1.3225
b05	-24.04%	-15.15%	1.4980	-25.90%	-16.32%	1.4870	-25.64%	-16.16%	1.4994
b06	-21.30%	-6.55%	0.4603	-26.85%	-8.26%	0.4323	-23.15%	-7.12%	0.4391
b07	-19.35%	-8.31%	0.9289	-21.25%	-9.13%	0.9258	-21.14%	-9.08%	0.9513
b08	-2.31%	-0.93%	0.5986	-4.36%	-1.77%	0.5998	-20.26%	-8.83%	0.6000
b09	-10.26%	-2.72%	0.8264	-10.26%	-2.72%	0.8264	-13.19%	-3.50%	0.8061
b10	-2.56%	-1.18%	0.8995	-3.58%	-1.65%	0.9009	-7.16%	-3.29%	0.8023
b11	-14.71%	-9.99%	0.6000	-12.26%	-8.33%	0.5998	-4.16%	-2.30%	0.6001
b12	-10.79%	-4.89%	0.6000	-12.84%	-5.82%	0.5996	-10.46%	-4.67%	0.6006
b13	-9.35%	-2.72%	0.7043	-9.52%	-2.77%	0.8060	-13.27%	-3.86%	0.7588
b14	-21.09%	-17.18%	1.9999	-20.52%	-16.71%	2.0000	-6.02%	-4.91%	2.0000
b14_1	-27.93%	-21.06%	1.9999	-29.04%	-21.89%	1.9999	+34.37%	+25.91%	2.0000
b15	-18.37%	-11.45%	1.9989	-18.70%	-11.66%	1.9999	-19.41%	-12.10%	1.9996
b15_1	-18.78%	-11.70%	1.9977	-19.21%	-11.97%	1.9995	-19.30%	-12.02%	1.9998
b17	-18.02%	-11.12%	2.0000	-18.38%	-11.34%	1.9999	-18.04%	-11.03%	2.0000
b17_1	-15.90%	-9.77%	2.0000	-17.95%	-11.03%	2.0000	-19.27%	-11.85%	1.9999
b18	-27.53%	-19.86%	4.0000	-27.83%	-20.07%	3.9999	-31.81%	-22.86%	4.0000
b18_1	-20.64%	-15.04%	3.9998	-31.15%	-22.46%	4.0000	-34.72%	-24.96%	4.0000
b19	-32.77%	-23.69%	4.0000	-32.77%	-23.69%	4.0000	-37.91%	-27.32%	4.0000
b19_1	-33.73%	-24.41%	4.0000	-34.87%	-25.22%	4.0000	-38.44%	-27.76%	4.0000
b20	-11.13%	-9.71%	2.0000	-12.18%	-10.63%	2.0000	-13.80%	-12.03%	2.0000
b20_1	-10.91%	-9.44%	2.0000	-11.52%	-9.97%	2.0000	-9.12%	-7.86%	2.0014
b21	-8.98%	-7.78%	2.0000	-10.51%	-9.11%	2.0000	-9.17%	-7.93%	2.0000
b21_1	-9.60%	-8.29%	2.0015	-10.27%	-8.87%	2.0022	-6.20%	-5.36%	2.0017
b22	-11.44%	-10.62%	2.0000	-12.37%	-11.43%	2.0000	-15.00%	-13.73%	2.0017
b22_1	-21.07%	-18.26%	2.0000	-22.08%	-19.09%	2.0000	-12.24%	-10.98%	2.0005
Average	-16.12%	-10.37%	-	-17.30%	-11.06%	-	-15.64%	-9.28%	-
Worst	-0.40%	-0.09%	-	-0.40%	-0.09%	-	+34.37%	+25.91%	-
Best	-33.73%	-24.41%	-	-34.87%	-25.22%	-	-38.44%	-27.76%	-

7.3 Manufacturability as a cost function

The ISCAS'85 benchmarks were mapped with commercial tool A using different libraries, with different regularity rules: reference, 2D-gridded and 1D-restricted. Commercial tool A is focused in area reduction keeping the timing constraints attained, and therefore ignores the proposed metric. Then, the proposed methodology using the proposed cost function remapped the gate-level netlists provided by commercial tool A, trying to maximize the *#GDW*. Differently from the experiments on Section 7.2, these experiments do not try to exploit libraries with different amount of logic gates, but libraries with different levels of regularity. The values used are the same as the Section 6.4: a wafer with 600 cm^2 of area, dies of $4cm^2$, and *dd*=0.025 defects/cm^2.

7.3.1 Libraries used for manufacturability experiments

The libraries descriptions are in Table 7.10. In the reference library, the layouts do not include any lithography consideration. Therefore, the cells of the reference library are smaller, since a higher compression can be done in the cell layouts, and the lithography printability is worse. The 2D-gridded library is composed of cells that are restricted to be designed on a regularly spaced grid, but can use two-dimensional features. Notice that the 2D-gridded library has cells with slightly larger area if compared to the reference library cells, but have a better printability. The 1D-restricted library is composed by litho-friendly cells that are restricted to use one-dimensional features (GÓMEZ; MOLL, 2010). Notice that the 1D-restricted cells have a much larger area if compared to the reference library cells, but have a much better printability. The cell layouts were not evaluated with any lithography simulation. In order to perform the experiments, the cells were assigned random values of *CHSci*, according to the level of regularity of the library, as seen in Table 7.10.

Table 7.10 Libraries used for manufacturing improvement experiments.

Name	# of cells	# of functions	Description	*CHSci*
Reference	266	49	Commercial cell library, no restrictions of features	20 to 30
2D-gridded	266	49	Same cells of reference library, with layout restricted to two dimensions (a grid) features	10 to 20
1D-restricted	266	49	Same cells of reference library, with layout restricted to one dimension features	5 to 10

7.3.2 ISCAS'85 benchmarks manufacturability results

Since foundry details on lithography resolution and yield are not known, four different values of *sld* were used, as can be seen in Tables 7.11, 7.12 and 7.13. For *sld* equals to 0.5, lithography resolution has less influence than other sources of yield loss. For *sld* equals to 1, density of defects and lithography printability are in the same range. For *sld* equals to 2, lithography resolution defects have more influence than critical area defects. Finally, for sld equals to 5, lithography is five times more important than critical area defects. Notice that mapping considering the cost function always improves the number of good dies per wafer (*#GDW*) when the proposed mapping is compared to an area-oriented mapping with same *sld*, and same library. This is verified in Tables 7.11, 7.12 and 7.13.

Results for the reference library are shown in Table 7.11. Notice that the number of dies per wafer (*#DW*) is always 150 for the reference library mapped by commercial tool A. This happens because the results were scaled to a number of instances of the benchmarks such that the total number of instances represents a die of 4 cm^2. This is done according to Equation (6.14) and Equation (6.15). All results of the experiment are scaled to have a number of instances equal to the reference benchmark (the number of instances vary from benchmark to benchmark, but it is constant for a single benchmark circuit).

Results for the 2D-gridded library are shown in Table 7.12 and results for the 1D-restricted library are shown in Table 7.13. The results show that the remapping of the results given by commercial tool A can lead to a significant increase in the *#GDW*, for different libraries with different regularity, and for different severity of lithography

resolution defects. The increase of *#GDW* happened due to: (1) increase of *#DW* (i.e. smaller benchmarks area), and (2) increase in total yield (i.e. smaller critical area and better printability). The results can also be used to verify the proportion of lithography resolution defects that justifies the use of a more regular layout. The results show that for *sld* equals to 5, the use of 1D characteristic in final layout is able to achieve better *#GDW* than a non-regular reference library, even with a very larger area, and consequently much smaller *#DW*. This comparison can be seen in Table 7.14, where the results of reference library and the 1D-restricted library are compared.

Table 7.11 Manufacturability results for reference library.

Benchmark	Commercial tool A					Proposed manufacturing remapping methodology							
	#DW	sld=0.5	sld=1	sld=2	sld=5	sld=0.5		sld=1		sld=2		sld=5	
		#GDW	#GDW	#GDW	#GDW	#DW	#GDW	#DW	#GDW	#DW	#GDW	#DW	#GDW
c1355	150	125	113	95	57	165	141	164	129	164	112	163	70
c1908	150	125	113	94	55	157	133	158	124	157	105	156	65
c1908a	150	125	114	95	57	155	131	156	122	155	103	155	64
c2670	150	125	113	94	55	153	129	153	119	153	101	154	62
c2670a	150	125	113	95	56	155	131	153	119	152	100	154	63
c3540	150	126	114	96	59	160	136	160	126	159	108	160	70
c3540a	150	126	114	96	58	162	138	162	128	161	110	161	71
c432	150	125	113	95	56	154	130	154	120	154	102	154	63
c499	150	125	113	95	57	151	127	151	117	150	98	151	60
c5315	150	126	114	97	59	170	146	169	135	173	121	169	76
c5315a	150	126	115	97	60	174	150	173	139	174	122	173	80
c6288	150	125	114	96	58	152	128	151	117	151	100	151	62
c7552	150	125	114	96	57	173	149	174	140	174	123	177	85
c880a	150	125	114	95	57	155	131	155	121	155	103	155	63

Table 7.12 Manufacturability results for 2D-gridded library.

Benchmark	Commercial tool A					Proposed manufacturing remapping methodology							
	#DW	sld=0.5	sld=1	sld=2	sld=5	sld=0.5		sld=1		sld=2		sld=5	
		#GDW	#GDW	#GDW	#GDW	#DW	#GDW	#DW	#GDW	#DW	#GDW	#DW	#GDW
c1355	140	119	111	99	71	155	135	155	128	155	116	155	85
c1908	140	119	111	99	70	145	125	151	125	151	113	152	84
c1908a	142	121	113	100	70	148	128	148	121	150	111	149	79
c2670	142	121	113	101	72	147	127	148	121	147	108	146	78
c2670a	138	117	109	97	68	142	122	142	115	143	105	142	75
c3540	142	122	114	102	74	152	132	152	125	153	114	151	82
c3540a	143	123	115	104	76	151	131	152	126	153	115	152	86
c432	136	116	109	97	71	143	123	147	121	147	110	145	80
c499	142	121	113	100	70	147	127	146	119	147	107	147	77
c5315	141	121	113	101	73	154	134	155	128	153	114	153	85
c5315a	142	122	114	102	74	157	137	157	130	156	117	156	87
c6288	135	114	106	94	65	138	118	136	109	137	98	137	69
c7552	142	122	114	102	73	165	145	162	135	162	124	163	95
c880a	139	119	111	99	71	146	126	146	119	146	108	147	80

Table 7.13 Manufacturability results for 1D-restricted library.

Benchmark	Commercial tool A					Proposed manufacturing remapping methodology							
	#DW	sld=0.5	sld=1	sld=2	sld=5	sld=0.5		sld=1		sld=2		sld=5	
		#GDW	#GDW	#GDW	#GDW	#DW	#GDW	#DW	#GDW	#DW	#GDW	#DW	#GDW
c1355	117	100	95	89	73	126	109	128	108	129	102	128	84
c1908	119	102	98	91	75	126	109	125	105	125	99	124	81
c1908a	117	100	96	89	73	121	104	121	101	121	95	124	81
c2670	117	100	96	89	74	121	105	122	102	123	96	122	80
c2670a	115	98	94	87	72	120	103	120	100	121	95	120	78
c3540	116	99	94	88	72	128	112	127	107	127	101	127	85
c3540a	116	99	94	88	72	124	107	124	104	124	98	124	82
c432	118	101	97	91	75	122	106	122	102	122	96	122	80
c499	117	100	96	89	73	117	100	117	97	117	91	117	74
c5315	116	99	94	88	71	130	114	130	110	130	104	130	88
c5315a	117	100	95	89	72	131	114	131	111	131	105	132	90
c6288	117	100	95	89	72	118	101	118	98	118	91	118	75
c7552	117	100	95	89	72	128	111	129	109	128	102	128	85
c880a	114	97	93	86	71	119	102	119	99	119	93	120	77

Table 7.144 Comparison between 1D-restricted library with reference library.

Benchmark	sld=0.5	sld=1	sld=2	sld=5
	#GDW	#GDW	#GDW	#GDW
c1355	-12.80%	-4.42%	+7.37%	+47.37%
c1908	-12.80%	-7.08%	+5.32%	+47.27%
c1908a	-16.80%	-11.40%	0.00%	+42.11%
c2670	-16.00%	-9.73%	+2.13%	+45.45%
c2670a	-17.60%	-11.50%	0.00%	+39.29%
c3540	-11.11%	-6.14%	+5.21%	+44.07%
c3540a	-15.08%	-8.77%	+2.08%	+41.38%
c432	-15.20%	-9.73%	+1.05%	+42.86%
c499	-20.00%	-14.16%	-4.21%	+29.82%
c5315	-9.52%	-3.51%	+7.22%	+49.15%
c5315a	-9.52%	-3.48%	+8.25%	+50.00%
c6288	-19.20%	-14.04%	-5.21%	+29.31%
c7552	-11.20%	-4.39%	+6.25%	+49.12%
c880a	-18.40%	-13.16%	-2.11%	+35.09%

8. CONCLUSIONS AND FUTURE WORK

The main contribution of this work was the introduction of the concept of *k*-cuts and *kl*-cuts performed on top of mapped circuits as opposed to computing *k*-cuts and *kl*-cuts on top of AIG representations. Besides bringing the idea from a technology independent data structure to a technology dependente gate netlist, three related contributions were also introduced: (1) algorithms to enumerate *k*-cuts and *kl*-cuts on top of mapped circuits; (2) a complete and operational remapping flow based on *kl*-cuts, which is able to reduce the area of circuits mapped by commercial logic synthesis tools; and (3) a novel manufacturing cost function to be used in the logic synthesis process, which considers lithography printability in order to increase the number of good dies per wafer manufactured.

The first contribution of this work is a comparison of *k*-cuts and *kl*-cuts performed on top of mapped circuits as opposed to computing *k*-cuts and *kl*-cuts on top of AIG representations. The main differences lie on (1) the number of inputs for the 2-input AND nodes used on AIGs and the nodes of a gate netlist which may have several inputs, and (2) the existence of explicit inverters and buffers, appearing as nodes, in the netlist compared to the use of negated or direct edges used in the AIG. Moreover, algorithms to enumerate *k*-cuts and *kl*-cuts on top of a netlist representation were proposed and implemented.

The second contribution presented is an iterative remapping flow, based on local transformations using *kl*-cuts. The proposed approach was implemented in an operational tool called KLever2, and it is able to reduce a cost function such as area, while respecting timing constraints. A complete suite of implementations and knownledge was necessary to implement such a tool: a Liberty parser and library data structure; a structural Verilog parser and mapped circuit data structure; an SDC parser and data structure; *k*-cut and *kl*-cut enumeration algorithms, parser and data structure; an extension of a multiple output P-signature algorithm, in order to consider *polarity don't cares*; an STA engine, with results comparable to a commercial tool; Boolean factoring aggressive algorithms; and logic tree mapping algorithms. For the benchmark circuits analyzed (ISCAS'85, ISCAS'89 and ITC'99), results show that the proposed flow can reduce combinational area in up to 38%, while still respecting the required timing. Also, the experiments have show that the use of complex logic gates is not well explored by commercial tools. The use of complex gates is better explored by the tool developed, showing a higher quality of results with a larger amount of different combinational cells. The proposed flow is composed of many heuristics. The quality of results is due to a combination of the following attributes: (1) use of *kl*-cuts which minimize the support of the Boolean functions; (2) extraction of full context, by using *kl*-cuts instead of *k*-cuts; (3) use of aggressive Boolean optimization techniques to optimize sub-circuits (*kl*-

cuts); and (4) allow only substitutions that improve area and do not impact negatively on the timing constraints.

The third contribution of this work was the introduction of a novel yield model for integrated circuits manufacturing, considering lithography printability and density of defects, which can be used as a cost function for logic synthesis process. A technology remapping tool using *kl*-cuts was developed considering this cost function, and results were compared with the results of a commercial logic synthesis tool for three different libraries, with different printability and area overhead characteristics. The proposed methodology establishes a new standpoint in the field of regular layout by introducing a metric to tradeoff area and printability of layouts. Many of the previous works completely ignored these tradeoffs and simply pointed out that regularity is expected to improve yield somehow, without presenting or discussing metrics. Therefore, a great contribution of this work is to propose a discussion about the tradeoffs between different area overheads and different levels of lithography printability for regular layouts. Another important contribution is to show that taking the proposed cost function into account during technology mapping produces circuits with larger number of good dies per wafer, when compared to simply minimizing the area.

8.1 Future work

Several improvements in the proposed approaches can be done in order to improve runtime and quality of results. For example, more types of *kl*-cuts could be explored, such as factor cuts (CHATTERJEE, 2006-b), which technique is able to find larger sub-circuits. Also, sub-circuits with higher amount of inputs could be found, if an approach such as priority cuts (MISHCHENKO, 2007) is applied. Larger sub-circuits tend to result in higher gains in the cost function desired, and a trade-off between runtime and quality of results can be done. Moreover, the runtime in larger circuits could be greatly improved, if a partial STA check is performed, such as the STA check proposed in (COUDERT, 1997). A timing fix engine would certainly improve a lot the quality of results and runtime, since the *kl*-cuts with a very large gain usually increase delay. By performing a timing fix, this large gain would be kept, and the *kl*-cuts that overlap with the *kl*-cut replaced would not be tested, and therefore less STA checks would be performed. Besides quality of results and runtime, different cost functions could be investigated, such as power (TIWARI, 1993), and a combination of cost functions.

REFERENCES

AITKEN, R. DFM metrics for standard cells. IN: INTERNATIONAL SYMPOSIUM ON QUALITY ELECTRONIC DESIGN, 2006…**Proceedings.** [S.l.:s.n.], 2006. p. 491-496.

Berkeley Logic Synthesis and Verification Group. **ABC**: A System for Sequential Synthesis and Verification. Accessed in November of 2012. Available at `http://www.eecs.berkeley.edu/~alanmi/abc/`.

BENINI, L.; VUILLOD, P.; DE MICHELI, G. Iterative remapping for logic circuits. **IEEE Trans. on Computer-Aided Design**, vol. 17, pp. 948–964, 1998.

BHASKER, J.; CHADHA, R. **Static Timing Analysis for Nanometer Designs: A Practical Approach.** [S.l.]: Springer, 2009.

BHATNAGAR, H. **Advanced ASIC Chip Synthesis Using Synopsys® Design Compiler® Physical Compiler® and PrimeTime®.** [S.l.]: Springer, 2001.

BRAYTON, R. K. Factoring logic functions. **IBM Journal of Research and Development**, vol. 31, no. 2, pp.187-98. Mar 1987.

BUBEL, I. et al. AFFCCA: A Tool for Critical Area Analysis with Circular Defects and Lithography Deformed Layout. IN: INTERNATIONAL WORKSHOP ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 1995…**Proceedings.** [S.l.:s.n.], 1995. p. 10-18.

CORREIA, V.; REIS, A.I. Advanced technology mapping for standard-cell generators. IN: 17TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2004…**Proceedings.** [S.l.:s.n.], 2004. p. 254-259.

COUDERT, O. Two-level logic minimization: an overview. **The VLSI journal Integration**, Elsevier, vol. 17, no. 2, pp. 97-140, 1994.

COUDERT, O. Gate sizing for constrained delay/power/area optimization. **IEEE Transactions on VLSI Systems**, vol. 5, no. 4, pp. 465- 472, 1997.

CHATTERJEE, S.; MISHCHENKO, A.; BRAYTON, R.; WANG, X.; KAM, T. Reducing structural bias in technology mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, vol. 25, n. 12, pp. 2894–2903, 2006.

CHATTERJEE, S.; MISHCHENKO, A.; BRAYTON, R. Factor cuts. IN: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2006...**Proceedings.** [S.l.:s.n.], 2006. p. 143–150.

DE MICHELI, G. **Synthesis and Optimization of Digital Circuits.** [S.l]: McGraw-Hill, New York, 1994.

DETJENS, E.; GANNOT, G.; RUDELL, R.; SANGIOVANNI-VINCENTELLI, A.; WANG, A. Technology mapping in MIS. IN: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1987...**Proceedings.** [S.l.:s.n.], 1987. p. 116–119.

DEY, S.; BRGLEZ, F.; KEDEM, G. Corolla based circuit partitioning and resynthesis. IN: 27TH DESIGN AUTOMATION CONFERENCE, 1990...**Proceedings.** [S.l.:s.n.], 1990. p. 607-612.

DING, D.; TORRES, A.J.; PIKUS, F.G.; PAN, D.Z. High performance lithographic hotspot detection using hierarchically refined machine learning. IN: 16TH ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2011...**Proceedings.** [S.l.:s.n.], 2011. p. 775-780.

FIŠER, P.; SCHMIDT, J. It Is Better to Run Iterative Resynthesis on Parts of the Circuit. IN: 19TH INTERNATIONAL WORKSHOP ON LOGIC & SYNTHESIS, 2010...**Proceedings.** [S.l.:s.n.], 2010. p. 17-24.

GÓMEZ, S.; MOLL, F. Lithography Aware Regular Cell Design Based on a Predictive Technology Model. **Journal of Low Power Electronics**, vol. 6, pp. 588-600, 2010.

HEINEKEN, H.T.; KHARE, J.; d'ABREU, M. Manufacturability Analysis of Standard Cell Libraries. IN: IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE, 1998...**Proceedings.** [S.l.:s.n.], 1998. p. 321-324.

HINSBERGER, U.; KOLLA, R. Boolean matching for large libraries. IN: 35TH ANNUAL DESIGN AUTOMATION CONFERENCE, 1998…**Proceedings.** [S.l.:s.n.], 1998. p. 206-211.

IWLS 2005 benchmarks. Accessed in November of 2012. Available at `http://iwls.org/iwls2005/benchmarks.html`.

KHETERPAL, V., ROVNER, V., HERSAN, T.G., MOTIANI, D., TAKEGAWA, Y., STROJWAS, A.J. PILEGGI, L. Design methodology for IC manufacturability based on regular logic-bricks. IN: 42ND DESIGN AUTOMATION CONFERENCE, 2005…**Proceedings.** [S.l.:s.n.], 2005. p. 353-358.

KEUTZER, K. DAGON: technology binding and local optimization by DAG matching. IN: 24TH DESIGN AUTOMATION CONFERENCE**,** 1987…**Proceedings.** [S.l.:s.n.], 1987. p. 341-347.

KOREN, I.; KOREN, Z. Defect tolerance in VLSI circuits: techniques and yield analysis. **Proceedings of the IEEE**, vol. 86, no.9, pp. 1819-1838, 1998.

KRAVETS, V.N.; KUDVA, P. Implicit enumeration of structural changes in circuit optimization. IN: 42ND DESIGN AUTOMATION CONFERENCE, 2004…**Proceedings.** [S.l.:s.n.], 2004. p. 438-441.

KUNZ, W.; STOFFEL, D. **Reasoning in Boolean Networks**, [S.l]: Springer, Boston, 1997.

LEHMAN, E. Logic decomposition during technology mapping. IN: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1995…**Proceedings.** [S.l.:s.n.], 1995. p. 264-271.

LEHMAN, E.; WATANABE, Y.; GRODSTEIN, J.; HARKNESS, H. Logic decomposition during technology mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, vol. 16, n. 8, pp. 813-834, 1997.

LUO, J.; SINHA, S.; SU, Q.; KAWA, J.; CHIANG, C. An IC Manufacturing Yield Model Considering Intra-Die Variations. IN: DESIGN AUTOMATION CONFERENCE, 2006…**Proceedings.** [S.l.:s.n.], 2006. p. 749-754.

MAILHOT, F.; DI MICHELI, G. Algorithms for technology mapping based on binary decision diagrams and on Boolean operations. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, vol. 12, n. 5, pp. 599-620, 1993.

MARTINELLO JR., O. **KL-cuts: a new approach for logic synthesis targeting multiple output blocks.** 2010. 85 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

MARTINS, M.G.A.. **Functional Composition and Applications.** 2012. 95 f. Dissertação (Mestrado em Microeletrônica) – Instituto de Informática, UFRGS, Porto Alegre.

MISHCHENKO , A.; BRAYTON, R. Scalable Logic Synthesis using a Simple Circuit Structure. IN: 15TH INTERNATIONAL WORKSHOP ON LOGIC & SYNTHESIS, 2006…**Proceedings.** [S.l.:s.n.], 2006. p. 15-22.

MISHCHENKO, A.; CHO, S.; CHATTERJEE, S.; BRAYTON, R. Combinational and sequential mapping with priority cuts. IN: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2007…**Proceedings.** [S.l.:s.n.], 2007. p. 354–361.

MOORE, G. E. Craming more components onto integrated circuits. **Electronics**, vol. 38, number 8, 1965.

NARAYAN, A.; JAIN, J.; FUJITA, M.; SANGIOVANNI-VINCENTELLI, A. Partitioned ROBDDs—a compact, canonical and efficiently manipulable representation for Boolean functions. IN: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1997…**Proceedings.** [S.l.:s.n.], 1997. p. 547–554.

NARDI, A.; SANGIOVANNI-VINCENTELLI, A. L. Logic synthesis for manufacturability. **IEEE Design & Test of Computers**, vol. 21, no. 3, pp. 192- 199, May-June 2004.

PAN, P.; LIN, C. A New Retiming-based Technology Mapping Algorithm for LUT-based FPGAs. IN: 6TH INTERNATIONAL SYMPOSIUM ON FPGAS, 1998…**Proceedings.** [S.l.:s.n.], 1998. p. 35-42.

SASAO, T. **Switching theory for logic synthesis.** [S.l.]: Kluwer Academic Publishers, 1999.

SAVOJ, H.; BRAYTON, R.K. The use of observability and external don't cares for the simplification of multi-level networks. IN: 27TH DESIGN AUTOMATION CONFERENCE, 1990…**Proceedings.** [S.l.:s.n.], 1990. p. 297-301.

SHAIK, S.A.; KHARE, J.; HEINEKEN, H.T. Manufacturability and testability oriented synthesis. IN: 13TH INTERNATIONAL CONFERENCE ON VLSI DESIGN, 2000…**Proceedings.** [S.l.:s.n.], 2000. p. 185-191.

SINGH, A.K.; MANI, M.; ORSHANSKY, M. Statistical technology mapping for parametric yield. IN: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2005…**Proceedings.** [S.l.:s.n.], 2005. p. 511-518.

STOK, L.; IYER, M. A.; SULLIVAN, A. J. Wavefront technology mapping. IN: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 1999…**Proceedings.** [S.l.:s.n.], 1999. p. 531-536.

SUNDARESWARAN, S.; MAZIASZ, R.; ROZENFELD, V.; SOTNIKOV, M.; KONSTANTIN, M. A sensitivity-aware methodology to improve cell layouts for DFM guidelines. IN: INTERNATIONAL SYMPOSIUM ON QUALITY ELECTRONIC DESIGN, 2011…**Proceedings.** [S.l.:s.n.], 2011. p. 1-6.

SZE, S.M. **Physics of semiconductor devices.** [S.l.]: Wiley, 2006.

TIWARI, V.; ASHAR, P.; MALIK, S. Technology Mapping for Low Power. IN: 30TH DESIGN AUTOMATION CONFERENCE, 1993…**Proceedings.** [S.l.:s.n.], 1993. p. 74-79.

WONG, B.; ZACH, F.; MOROZ, V.; MITTAL, A.; STARR, G.; KAHNG. A. **Nano CMOS Design for Manufacturability: Robust Circuit and Physical Design for Sub-65 nm Technology Nodes**, [S.l.]: John Wiley & Sons, 2009.

WUU, J.Y.; PIKUS, F.G.; TORRES, A.J.; MAREK-SADOWSKA, M. Detecting Context Sensitive Hot Spots in Standard Cell Libraries. IN: INTERNATIONAL SOCIETY FOR OPTICAL ENGINEERING SYMPOSIUM, 2009…**Proceedings.** [S.l.:s.n.], 2009. p. 727515-1 to 727515-9.