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

LEOMAR SOARES DA ROSA JUNIOR

Automatic Generation and Evaluation of Transistor Networks in Different Logic Styles

Thesis presented in partial fulfillment of the requirements for the degree of Doctor in Microelectronics.

Prof. Dr. André I. Reis
Advisor

Prof. Dr. Renato P. Ribas
Co-advisor

Porto Alegre, July 2008.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rosa Junior, Leomar Soares da

Automatic Generation and Evalution of Transistor Networks in Different Logic Styles / Leomar Soares da Rosa Junior – Porto Alegre: Programa de Pós-Graduação em Microeletrônica, 2008.

147 f.: il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica. Porto Alegre, BR – RS, 2008. Advisor: André I. Reis; Co-advisor: Renato P. Ribas.

1.Redes de transistores. 2.Células lógicas 3.Mapeamento tecnológico. 4.Teoria de chaves 5.Estilos lógicos 6.Estimativas de células I. Reis, André Inácio. II. Ribas, Renato Perez. III. Título.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my parents, Leomar Soares da Rosa and Maria Cecília Machado da Rosa, for their patience, encouragement and love. They were always supporting me and encouraging me with their best wishes.

I would also like to thank my aunt, Gilda Maria da Silva Machado, for her support. All this time that I was far from home she treated me as her son.

I acknowledge my fiancée, Pamela Bilhafan Disconzi. Although we were some kilometers apart, I felt we never were. She shared my all ups and downs over the phone and email and stood by me.

I would like to thank CAPES and Nangate for financial support. And also thank all my colleagues from Nangate-UFRGS Research Lab. They were always willing to help and give their best suggestions.

Finally, I would like to express my sincere gratitude to my advisors, André Inácio Reis and Renato Perez Ribas, for their brilliant guidance and their patience. They were always close and ready to guide me when the things seemed dark and the project seemed endless.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ASIC	Application-Specific Integrated Circuits
BDD	Binary Decision Diagram
CAD	Computer Aided Design
CMOS	Complementary Metal Oxide Semiconductor
CSP	Complementary Series-Parallel
CUDD	Colorado University Decision Diagram
FPGA	Field-Programmable Gate-Arrays
IC	Integrated Circuit
LBBDD	Lower Bound Binary Decision Diagram
LO	Larger Order
MOS	Metal Oxide Semiconductor
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
NCSP	Non-Complementary Series-Parallel
NO	Not Smaller and Larger Order
OpBDD	Optimized Binary Decision Diagram
POS	Product-Of-Sums
PTL	Pass Transistor Logic
PTM	Predictive Technology Model
ROBDD	Reduced and Ordered Binary Decision Diagram
SO	Smaller Order
SoC	System-On-Chip
SOP	Sum-Of-Products
SP-BDD	Series-Parallel Binary Decision Diagram
TBDD	Terminal Suppressed Binary Decision Diagram
TM-BDD	Transistor Mapped Binary Decision Diagram
UFRGS	Universidade Federal do Rio Grande do Sul
ULSI	Ultra Large Scale Integration
VLSI	Very Large Scale Integration

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Currently, VLSI design has established a dominant role in the electronics industry. Automated tools have enabled designers to manipulate more transistors on a design project and shorten the design cycle. In particular, logic synthesis tools have contributed significantly to reduce the design cycle time. In full-custom designs, manual generation of transistor netlists for each functional block is performed, but this is an extremely time-consuming task. In this sense, it becomes comfortable to have efficient algorithms to derive transistor networks automatically. There are several kinds of transistor networks arrangements. These different networks present different behaviors in terms of area, delay and power consumption. Thus, not only automatic transistor networks generation is important, but also an automated technique to evaluate and to compare the distinct switch networks is fundamental to guide designers that need to achieve efficient circuit implementations. This evaluation not necessarily needs to be an expensive electrical characterization process. It can be obtained through estimation processes capable of delivering good information about the logic cells behavior. This idea is useful for those designers that desire to generate and to evaluate potential transistor network implementations to feed standard-cell flow designs (using cell libraries), or for those designers who target the use of library-free technology mapping concept (using automatic cells generators). In this context, this work presents an automated transistor network generator able to delivery different kinds of networks in several logic styles. In order to compare the obtained networks, some estimation techniques are employed. A comparison is done over a set of Boolean function benchmarks, showing the advantages of using alternative logic styles over the traditional Complementary Series-Parallel CMOS (CSP CMOS).

Keywords: Transistor Networks, Logic Cells, Technology Mapping, Switch Theory, CMOS Logic Styles.

Geração Automática e Avaliação de Redes de Transistores em Diferentes Estilos Lógicos

RESUMO

O projeto e o desenvolvimento de circuitos integrados é um dos mais importantes e aquecidos segmentos da indústria eletrônica da atualidade. Neste cenário, ferramentas de automação têm possibilitado aos projetistas manipular uma elevada quantidade de transistores em circuitos cada vez mais complexos, diminuindo, assim, o tempo de projeto. Em especial, ferramentas de síntese lógica têm contribuído significativamente para reduzir o ciclo de desenvolvimento. Na metodologia de projeto *full-custom*, cada bloco funcional tem sua geração realizada de forma manual, desde a implementação das redes de transistores até a geração do leiaute. Entretanto, esta tarefa é extremamente custosa em tempo de projeto. Neste contexto, torna-se confortável ter a disposição algoritmos dedicados para derivar redes de transistores automaticamente. Diversos tipos de arranjos de transistores são encontrados na literatura. Estas diferentes redes de transistores apresentam diferentes comportamentos em termos de consumo de área, consumo de potência e velocidade. Desta forma, não apenas a geração automática de redes de transistores é importante, mas também técnicas automatizadas para avaliar e comparar estas distintas redes de chaves é de fundamental importância para guiar o projetista que deseja alcançar implementações de circuitos eficientes. Estas avaliações não precisam ser necessariamente processos custosos de caracterização elétrica. Elas podem ser realizadas através de estimativas capazes de fornecer informações acuradas sobre o comportamento das redes. Esta idéia pode ser utilizada por projetistas que desejam gerar e avaliar potenciais soluções em redes de transistores para alimentar fluxos *standard-cell* (utilizando bibliotecas de células), ou por aqueles que utilizam a abordagem de mapeamento tecnológico *library-free* (fazendo uso de geradores de células). Neste contexto, este trabalho apresenta um gerador automático de redes de transistores capaz de fornecer diferentes tipos de redes em diversos estilos lógicos. Para comparar as redes geradas, algumas técnicas de estimativa são empregadas. Comparações são realizadas sobre conjuntos distintos de funções Booleanas, demonstrando as vantagens da utilização de lógicas alternativas em relação ao difundido padrão CMOS.

Palavras-chave: Redes de Transistores, Células Lógicas, Mapeamento Tecnológico, Teoria de Chaves, Estilos Lógicos CMOS.

1 INTRODUCTION

Microelectronics became the key technology of many industry branches like information technology, telecommunication, medical equipment and consumer electronics. The ability of microelectronics to process, transport and store data digitally made many new applications possible. The continuously increasing level of integration of electronic devices on a single substrate has led to the fabrication of increasingly complex systems. An **Integrated Circuit** (IC) is an electronic system consisting of a number of miniaturized electronic devices, such as transistors, resistors, capacitors and inductors, built on a monolithic semiconductor substrate. The large majority of the current ICs are implemented in the Metal-Oxide-Semiconductor (MOS) technology (WESTE, 2005; RABAEY, 2003).

The IC design can be divided into two broad categories: analog and digital design. **Analog design** is used in the development of operational amplifiers, linear regulators, phase-locked loops, oscillators and active filters. Analog design is more concerned with the physics of the semiconductor devices such as gain, matching, power dissipation, and resistance. Fidelity of analog signal amplification and filtering is usually critical and as a result, analog ICs use larger area active devices than digital designs and are usually less dense in circuitry. In the other hand, **digital IC design** is used to produce components such as microprocessors, FPGAs (Field-Programmable Gate-Arrays), memories and digital ASICs (Application-Specific Integrated Circuits). Digital design focuses on logical correctness, maximizing circuit density, and placing circuits so that clock and timing signals are routed efficiently.

Since the advent of the technology for constructing ICs, integration density and performance of these electronic systems have gone through an astounding revolution driven by the ability of integrating in a single system more and more transistors, the devices responsible by most of the complexity of digital ICs. Indeed, the increase in the number of transistors that can be integrated in a single die has grown exponentially in the last three decades, as predicted by the so called **Moore's Law** (INTEL, 2007; MOORE, 1965). Figure 1.1 illustrates how this increase prediction has been proved correct so far. Although it has been frequently stated that such increase might cease in a few years due to physical limitations of IC manufacturing technologies, new design

methodologies and fabrication process breakthroughs have proven that such cease can be postponed (MOORE, 2003).

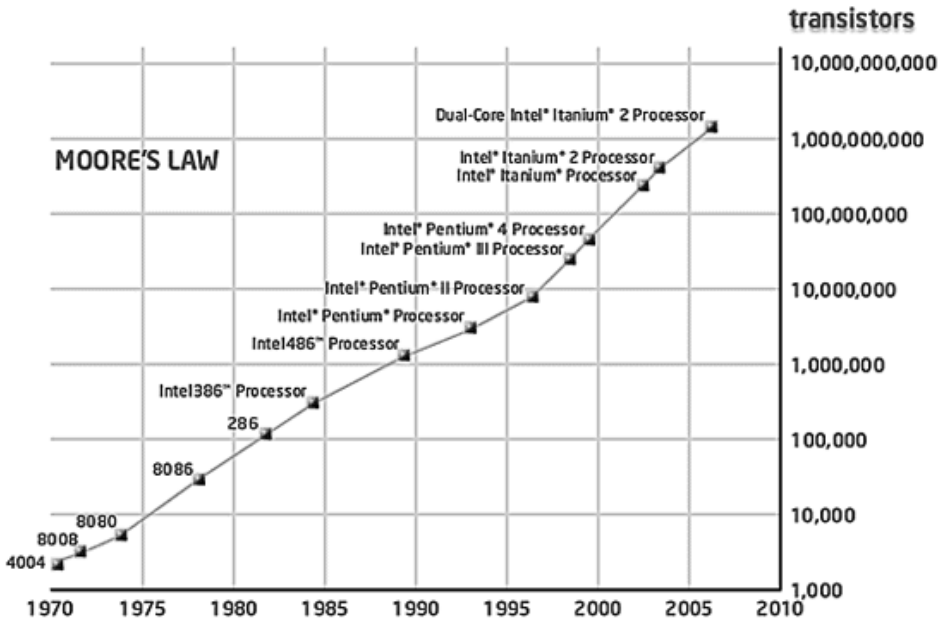


Figure 1.1: Moore's Law graph showing the exponential increase in the number of transistors along the last three decades for the microprocessors family from Intel (INTEL, 2007).

Essentially, there are two main flows when designing digital ICs that lead to two contrasting situations: fast design and high-performance design. Fast design here means short time-to-market; for this kind of approach for IC design, a **standard-cell design methodology** is the most commonly used approach. On the other hand, design for high-performance uses a **full-custom design methodology**, as this kind of design is completely customized to the high performance in terms of area, speed and power consumption (DEMICHELI, 1994).

In full-custom design the logic and physical synthesis attain usually the highest performance and smallest size, making use of the most advanced technologies (CHEN, 2000). It is the most technology dependent design approach, since each switch element present in every cell is manually fine-tuned in order to explore all the performance advantages that a given technology can deliver. The benefits of full-custom design in general include reduced area, performance improvements and also the ability to integrate analog components and other pre-designed components such as microprocessor cores that form a System-on-Chip (SoC). The disadvantages of full-custom can include increased manufacturing and design time, and much higher skill requirements on the part of the design team.

The proposal of standard-cell design is to reduce the implementation effort by reusing a library of cells. The advantage of this approach is that the cells only need to be

designed and verified once for a given technology, and they can be reused many times, thus amortizing the design cost. The disadvantage is that the constrained nature of the library, especially due to the limited number of cells, reduces the possibility of fine-tuning the design (RABAEY, 2005). According to Scott (1994), the quality of a synthesized design based on standard-cells depends on three main components: (a) the synthesis tool, (b) the place and route tools, and (c) the target cell library. Choosing the right cell library may have a significant impact on the characteristics of a circuit (VUJKOVIC, 2002; SECHEN, 2003).

Cell library is a finite set of logic cells that implements different Boolean functions with different drive strengths and topologies. Traditionally, the technology mapping methods rely on static pre-characterized libraries aiming delay, area and power optimizations. Each cell in the library is fully characterized through many simulations, resulting in a set of accurate information about the behavior of the cell. According to Sechen (2003), the design and characterization costs of a library are expensive. Therefore, commercial libraries are typically composed of few hundred combinational cells and sequential elements (latches and flip-flops) for which layouts have been optimized for a particular technology. As a result, designers are restricted to use these cells in their circuits. An example of a well-known and widely used academic library is presented in Appendix A.

Technology mapping is the procedure of expressing a given Boolean network in terms of logic cells or gates. Typically, the objective function aims the optimal use of all gates in the library to implement a circuit with critical-path delay less than a target value and minimum area. The most existing techniques for technology mapping are based on pre-characterized cell libraries (KEUTZER, 1987; KUKIMOTO, 1998; STOK, 1999; MISCHENKO, 2005). These techniques are also known as **library-based methodology**. Ideally, technology mapping algorithms and tools should be able to satisfy several goals and to handle different libraries. It is a quite hard task since the cell libraries normally have a different set of cells that implements a limited set of logic functions. A library of fixed size restricts the choices for covering a given circuit. Figure 1.2 shows the typical design flow considering technology mapping methodologies based on libraries with a fixed size.

Some works in the literature try to optimize logic cells on specific circuits and implementations. Typical optimizations have been limited to the design of buffers and inverter chains, implemented to minimize power consumption (MA, 1994) and delay (VEMURA, 1991; PRUNTY, 1992). Other ones try to optimize logic cells from existing cell libraries in order to adjust them to the circuit requirements (FISHER, 1996). Recent researches advocate that transistor-level optimizations are a powerful technique to improve the circuit performance (PANDA, 1998; BHATTACHARYA, 2002; YOSHIDA, 2006). In Roy (2005) some parts of the circuit are removed and replaced by optimized cells to attain the technical specification. This replacement is done as a post-processing step, after the circuit has been defined by the technology mapping task. In this strategy, the final circuit is composed by two types of cells,

derived from commercial library container and handcrafted complex gates. Figure 1.3 illustrates this idea, indicating a considerable propagation delay gain for the circuit.

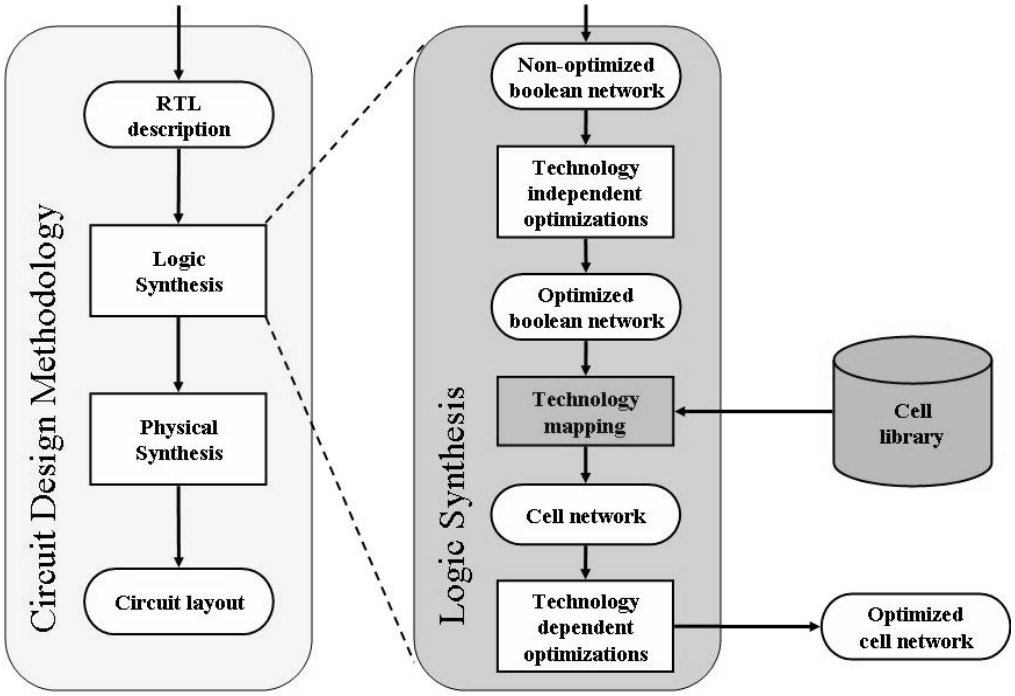
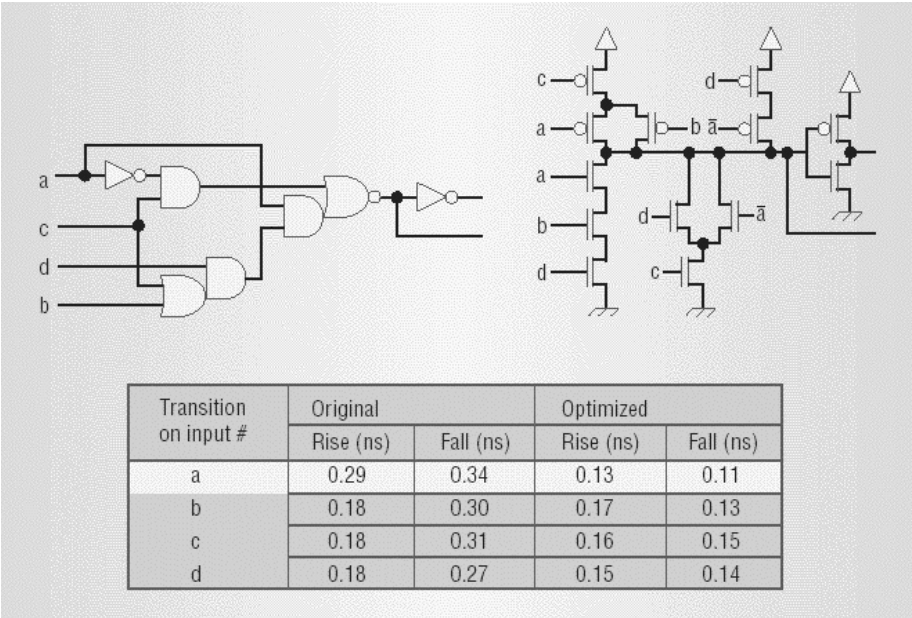


Figure 1.2: Digital circuit design methodology using predefined cell library (MARQUES, 2007).



Transition on input #	Original		Optimized	
	Rise (ns)	Fall (ns)	Rise (ns)	Fall (ns)
a	0.29	0.34	0.13	0.11
b	0.18	0.30	0.17	0.13
c	0.18	0.31	0.16	0.15
d	0.18	0.27	0.15	0.14

Figure 1.3: Circuit optimization using complex gates (ROY, 2005).

Usually, cell libraries are composed of a few tens of logic cells, due to the engineering effort to design and characterize each one. These cells have been previously

tested and validated, and all information about their behavior is described in a database which, in turn, is used during the technology mapping procedure.

Some researchers have observed that large cell libraries could lead to a better circuit implementation (VUJKOVIC, 2002). However, the number of potential logic function increases exponentially with the number of inputs. Therefore, it is not possible to characterize and implement all existing functions in a huge library. The processes of electrical characterization and layout generation are extremely computing demand, making the possibility of having large cell libraries unfeasible (SECHEN, 1996).

Other approaches for technology mapping propose techniques based on automatic cell generators. These approaches are known as **library-free** (BERKELAAR, 1988; REIS, 1998; STOK, 1999; JIANG, 2001; CORREIA, 2004; MARQUES, 2007). Instead of having a predefined static library, they assume that arbitrary cells can be generated on-the-fly through a cell generator, increasing the matching search space. The mapping algorithm defines the set of cells required in the circuit implementation, and this **virtual library** is used as input for a cell generator which provides the logic cell layouts that are further used in the physical synthesis. Figure 1.4 illustrates the logic synthesis flow of this approach.

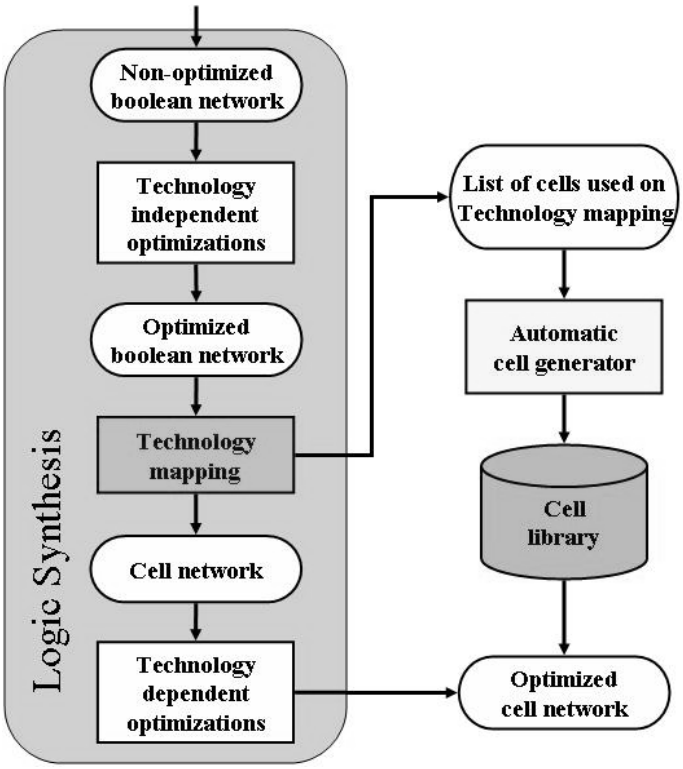


Figure 1.4: Digital circuit design methodology using virtual library (MARQUES, 2007).

Notice that, the quality of mapped circuits is highly dependent on the richness of the library in terms of the number of implemented logic functions, drive strengths

and topologies. Libraries that implements a large number of Boolean functions leads to better results when compared to sparsely populated libraries. In Keutzer (1987) the impact of a library size was investigated. In this work it was demonstrated that a better area optimization can be achieved using large libraries. As Jiang (2001) has observed, the most recent device technologies encourage the usage of complex gates in deep-submicron circuits. It leads to better circuit performance. But, the main barrier for virtual library approach is the dependency of a good layout generator and the lack of accurate information about the cell behavior. Due to this, static pre-characterized libraries are still popular in the industry. This work addresses two problems associated with this flow. The first one is to find good quality transistor networks to implement the cells in the library. The second one is to use fast models to estimate cell area, timing and power on-the-fly.

1.1 Proposal of this Thesis

According to the previously statements, this work addresses the digital cell implementation and optimization at the transistor cell level. It is known that different logic styles result in transistor networks with different electrical and physical behavior. Although several transistor network styles are available, the standard-cell industry keeps using the standard CMOS. The library-free approach is a promising solution, but it presents the disadvantage of lacking the characterization information. The characterization process is expensive in terms of CPU, making impracticable the use of this technique to generate and evaluate cells on-the-fly. An alternative is the use of estimation techniques. By using fast and efficient methods to obtain estimative about the logic cells behavior, it is possible to generate library cells considering these estimated information as costs, avoiding the characterization process.

The estimation approach, adopted as solution in this work, not only can be used to feed library-free technology mapping flow, but also as a method to generate information about the behavior of cells to compose library containers. Thus, it is possible to generate specific libraries composed of cells with estimated costs regarding area, timing and power. These libraries are suitable to be used in traditional standard-cells design flow. The circuits can be mapped, tested and simulated. Once they meet the design constraints, them the designer can effectively implement the layout of the cells to obtain the final circuit. Commercial layout generators are available in the market, like the Nangate Library Creator, which accepts Spice netlist description as input to automatically generate the cell layout (NANGATE, 2008).

In this sense, this thesis proposes an automated flow for generating transistor cell networks in different logic styles and a technique to obtain information about the behavior of these cells through estimation methods. Furthermore, scientific contributions of this thesis are also:

- A new BDD-based transistor network logic style that respects the minimum number of switches in series to implement a given logic function;
- A factorization algorithm to optimize logic expressions and electrical networks;
- CAD tools for logic synthesis of Boolean functions, as well as for automatic generation and evaluation of transistor networks.

2 LOGIC SYNTHESIS AND SWITCH NETWORKS

Integrated circuits design presents a set of concepts and terminologies very specific and necessary for the understanding of the field. More specifically, logic synthesis definitions must be reviewed in order to permit the whole understanding of this work. The goal of this chapter is to present the conceptual framework on top of which work is built. This chapter is organized as follows. Firstly, this chapter introduces these concepts that will be used in following chapters. Secondly, this chapter presents a brief discussion about the history of switch theory and about logic switches. Finally, it discusses possible optimizations performed at the logic level, presenting a factorization method to achieve minimum literal Boolean expressions and a new kind of transistor network derived from BDD. For the following chapters, it is assumed that the reader has the knowledge of definitions described herein.

2.1 Basic Concepts and Terminology

The **Boolean set** B is defined as a two element set, $B = \{0, 1\}$, whose elements are interpreted as **logic values**, typically *'0' = false* and *'1' = true*. An n-dimensional Boolean set B^n is composed of all the distinct Boolean vectors of length *'n'*. For instance $B^0=\{\varnothing\}$, $B^1=B=\{0, 1\}$, $B^2=\{00, 01, 10, 11\}$ and $B3=\{000, 001, 010, 011, 100, 101, 110, 111\}$. It is easy to observe that B^n has 2^n elements. A **Boolean function** describes how to determine a Boolean value output based on some logic calculation from Boolean input vectors of length *'n'*. A **Boolean function** is a function of the form $f: B^n \rightarrow B$, where $B = \{0, 1\}$ is the Boolean domain and where *'n'* is a non-negative integer. A Boolean function $f: B^n \rightarrow B$ can be viewed as a function whose domain is composed of the set of all n-bit Boolean vectors (that means B^n, which contains 2^n elements) and whose image is composed of unidimensional Boolean vectors (i.e. $B^1=B$, which contains two elements). So every distinct n-input Boolean vector of B^n can point to a distinct one dimensional Boolean vector. This way, a function $f: B^n \rightarrow B$ has 2^n input positions pointing to a fixed value from B. As changing the value pointed by a single input vector changes the logic function, there are 2^{2^n} such functions, as the

output has 2^n positions that can be associated to two distinct values from B. In the case where $n = 0$, the function is simply a constant element of B. Boolean functions are also called **logic functions**.

Boolean variables are variables defined in the Boolean domain and generally assigned using alphanumeric characters. Examples of Boolean variables are: *a*, *b*, *c*, *x0*, *x1*, *y2*; if they are defined over the Boolean set. Boolean variable can assume arbitrary values in the Boolean domain *B*, i.e. Boolean variables can assume the values *'0'* or *'1'*.

There are three basic **Boolean operators**: *AND* ("*"), *OR* ("+") and *NOT* ("!"), which can be applied to Boolean values or functions. *AND* operator returns one (or true) when all the operands are true and returns false for the other cases. *OR* operator returns zero (or false) when all the operands are false and returns true otherwise. *AND* and *OR* operators are binary operators, as they require at least two elements to perform the operation. *NOT* operator, also called inversion or negation operator, is unary and can be applied to one element alone. *NOT* operator returns zero when the operand is one and *vice-versa*. The operands may be Boolean functions or Boolean constants.

Phase or **polarity** of a Boolean variable indicates if it is used in its direct or inverted form. Positive phase specifies the use of a variable without inversion, while negative phase specifies the use of its complement. A variable in its negative phase is noticed by the anteriority of a *NOT* operator ('!') as, for instance, *!a*, *!t*, etc. **Literal** is an instance of a Boolean variable in its positive or negative phase. Examples of literals are: *a*, *!a*, *,x0*, *!y2*. Notice that *a* and *x0* are positive literals, while *!a* and *!y2* are negative literals.

Input vector is an element that indicates the value of each Boolean variable in a given Boolean function. For a certain number of variables there is 2^n input vectors, where *n* is the total number of Boolean variables.

Boolean expressions or **Boolean equations** are representations of a Boolean function. Each Boolean function is distinct, as it represents just one association $f : B_n \rightarrow B$. However, it is possible to write a Boolean function in different forms using Boolean operators. For example, the two following Boolean equations represent the same Boolean function:

$$Eq1 = a * (b * c) + d * (e + c) \qquad (2.1)$$

$$Eq2 = c * (a * b + d) + d * e \qquad (2.2)$$

Boolean functions can also be represented in tabular form known as **truth table**. In a **truth table** representation, the output values are shown according to all possible input combinations. In other words, the truth table is a representation form where all function values are specified for all domain function. The truth table can be built for any number of input variables. However, all possible combinations for these input variables must be present. It means that each line of the truth table represents an input vector and its respective output value. Table 2.1 illustrates truth tables for basic

Boolean functions with two inputs *A* and *B*. Notice that those are functions defined as $B^2 \rightarrow B$; which means that the input vectors *[A,B]* can assume any of the four (2^2=4) values in B^2=*{00, 01, 10, 11}*. The *AND* and *OR* operators were already defined above. The operator *XOR* returns *'1'* when an odd number of inputs are equal to *'1'*. The operators *NAND*, *NOR* and *XNOR* are the inverted versions of *AND*, *OR* and *XOR,* respectively.

Table 2.1: Truth table for the 2-input basic functions.

A	**B**	***AND***	***OR***	***XOR***	***NAND***	***NOR***	***XNOR***
0	0	0	0	0	1	1	1
0	1	0	1	1	1	0	0
1	0	0	1	1	1	0	0
1	1	1	1	0	0	0	1

For a given Boolean function, the set of input vectors that produces an output value *'1'* is called **on-set**. In the same way, the set of input vectors that produces an output value *'0'* is called **off-set**.

A product of literals is an *AND* logic operation between these literals. *(a*b*c*e)* and *(!a*c*!d)* are examples of products. The **sum-of-products** (**SOP**) representation is the Boolean equation composed of *OR* logic operation in between two or more products. The following equations are examples of SOP:

$$Eq3 = !a * b * !c * d + a * b * !c * !d + !a * !b * c * d \quad (2.3)$$

$$Eq4 = x0 * !x1 * x2 + x1 * x2 * !x3 + !x0 * x1 * !x2 * x3 \quad (2.4)$$

$$Eq5 = (a * b * c) + (!a * c * d) + (b * !c * !d) \quad (2.5)$$

There is a straightforward manner to derive a SOP representation from a truth table. To do that, it is only necessary to extract all lines (products), that present output values *one* in the truth table, and to implement *OR* operations between these products. Such equation is known as a Boolean equation in the **SOP canonical form**. Canonical forms have this name because they preserve a one-to-one relation with the truth table, meaning that there is only one canonical SOP per Boolean function, even if many different non-canonical equations can exist. Some of the non-canonical equations can present a reduced number of literals compared to canonical SOPs. As a consequence, a canonical SOP is not necessarily the minimal representation for most Boolean functions. The procedure of building a SOP with minimum number of literals is more elaborate, and can be done with algorithms like Quine-McCluskey (QUINE, 1955; MCCLUSKEY, 1956). It is important to notice that all variables must be present in each product to guarantee that the equation is in the canonical form. Moreover, it cannot

contain repeated products. An example of equation in canonical form is the equation (2.3).

The **product-of-sums** (**POS**) representation is very similar to the sum-of-products one. The difference is that the Boolean equation is composed of *AND* logic operation in between two or more sums of literals. Also, to build the sums, all lines that present output value *'0'* in the truth table are considered. Notice that, similar to SOP, all variables must be present in each sum to guarantee the POS in the canonical form. Again, a canonical POS is not necessarily the minimal representation of Boolean functions.

A product containing all variables that compose the function is called **minterm**. A minterm keeps a unique relation with just one line of the truth table. The Table 2.2 illustrates a truth table for a 3-input function and the minterms for each line.

Table 2.2: Relation between minterms and lines of the truth table.

A	B	C	Minterm	Equation
0	0	0	m0	!A*!B*!C
0	0	1	m1	!A*!B*C
0	1	0	m2	!A*B*!C
0	1	1	m3	!A*B*C
1	0	0	m4	A*!B*!C
1	0	1	m5	A*!B*C
1	1	0	m6	A*B*!C
1	1	1	m7	A*B*C

Implicant minterms are all minterms whose the function value is equal to *'1'*. Thus, as mentioned before, a canonical SOP is the one composed of all implicant minterms of a given logic function.

A sum containing all variables that compose the function is called **maxterm**. A maxterm also keeps a unique relation with just one line of the truth table. The Table 2.3 illustrates a truth table for a 3-input function and the maxterms for each line.

Cube is a set of minterms. While a minterm presents a relation with just one line of the truth table, a cube presents a relation with one line or a set of lines of the truth table. For instance, considering the two minterms *(!a*b*c*!d)* and *(!a*b*c*d)*, that compose the equation f = *(!a*b*c*!d) + (!a*b*c*d)*, it is possible to group them through equation manipulations, as follow:

$$(!a*b*c*!d) + (!a*b*c*d) = (!a*b*c) * (!d + d) = (!a*b*c) * 1 = (!a*b*c) \qquad (2.6)$$

This new simplified product *(!a*b*c)*, derived from the two given minterms, is called a cube. When a cube is only composed of implicant minterms, this cube is called an **implicant cube**.

Table 2.3: Relation between maxterms and lines of the truth table.

A	B	C	Maxterm	Equation
0	0	0	M0	A+B+C
0	0	1	M1	A+B+!C
0	1	0	M2	A+!B+C
0	1	1	M3	A+!B+!C
1	0	0	M4	!A+B+C
1	0	1	M5	!A+B+!C
1	1	0	M6	!A+!B+C
1	1	1	M7	!A+!B+!C

The **Karnaugh map** representation is an indexed matrix that permits to identify the adjacent minterms. Figure 2.1 illustrates a 4-input Karnaugh map for the minterms *(!a*b*c*!d)* and *(!a*b*c*d)*. In this example, the values in the columns represent the logic values for variables *'a'* and *'b'*, while the values in the lines represent the logic values for variables *c* and *d*.

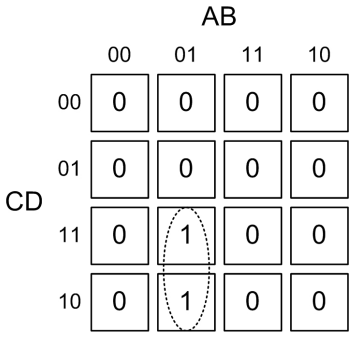


Figure 2.1: Karnaugh map illustration.

As shown in the example, it is possible to group the two adjacent minterms to obtain a cube. When a cube cannot be grouped with any other cube or existing minterm, in order to form a larger cube, then this cube is called a **prime cube**.

When grouping adjacent minterms to compose cubes, some important definitions become apparent. The first one is related to the **cube literal cost** of a SOP. The cube literal cost of a SOP is the maximum number of literals in a single cube of the SOP. Consider the function given by the following prime irredundant SOP.

$$f = !a*!b*!d + !a*b*!c + a*!d*!e + a*c*d + b*c*!d*e \qquad (2.7)$$

The cube literal cost of this SOP is four, as it has cubes with up to four literals.

The second definition is related to the **prime irredundant SOP with minimum cube literal cost** (SCHNEIDER, 2007). A prime irredundant SOP with minimum cube literal cost for function *f* is a prime irredundant SOP where the maximum number of literals in a single cube is minimum for function *f*. Consider the function given by the following prime irredundant SOP.

$$f = !a*!b*!d + !a*b*!c + a*!d*!e + a*c*d + !a*!d*e + a*b*c \qquad (2.8)$$

The cube literal cost of this SOP is three, as it has only cubes with three literals. The prime irredundant SOPs given by equations (2.7) and (2.8) represent the same logic function. It is possible to show that the SOP in equation (2.8) is a prime irredundant SOP with minimum cube literal cost for function *f*, as no solution containing cubes with at most two literals is possible for *f*.

Consider now, as an example, the function *f* given by equations (2.7) and (2.8). The cubes *!a*b*!c* and *a*c*d* are **essential primes**, the remaining cubes and minterms are shown in the **covering table** of Table 2.4. It is possible to see that the cube *b*c*!d*e*, with four literals, would be chosen in a **minimum literal cost SOP solution** like that presented in equation (2.7). However, this cube can be deleted from the covering table, leading to the **minimum cube literal cost SOP** presented in equation (2.8). The deletion of cubes with three literals would lead to an unfeasible covering table, as no minterm could be covered.

Table 2.4: Covering table for function *f*.

	minterms									
cubes	0	1	4	5	13	16	20	24	28	29
!a*!c*!d	●	●								
!c*!d*!e	●					●		●		
!b*!d*!e	●		●			●	●			
!a*!b*!d	●	●	●	●						
!a*!d*e		●		●	●					
b*c*!d*e					●					●
a*b*c									●	●
a*c*!e							●		●	
a*!d*!e						●	●	●	●	

As mentioned before, for a given number of input variables there is a well-defined **number of functions**. This number is given by 2^{2^n}, where *'n'* is the number of input variables (SASAO, 2000). According to this statement, the number of 2-input functions is 16, 3-input functions is 256, 4-input functions is 65,536, 5-input functions is 4,294,967,296, and so on. This exponential relation lead to a search space almost

intractable if many operations need to be repeated in a set of functions with more than 4-input. The set of *n*-input functions can be classified into different classes (set of functions) for different reasons: one is to reduce the search space, other is to group functions with equivalent or similar implementations. These sets are known as **equivalence classes**, and they may be obtained through input permutation/inversion as well as output inversion. P-class, N(in)-class, N(out)-class, NP-class, PN-class, and NPN-class are the possible reduced sets (SASAO, 2000; CORREIA, 2001). A class is a subset of logically equivalent functions as a result of a specific operation or their combination.

The first possible operation to obtain equivalent functions is the permutation of inputs. Table 2.5 presents an example of that operation. Notice that the input vectors are ordered differently for the truth tables of *f2* (ABC ordering) and *f4* (BCA ordering). The two functions are equivalent as once the permutation of inputs is done the truth tables are identical. Thus, *f2* and *f4* are equivalent by permutation, and can be gathered in a P-class set. The second operation to achieve equivalent functions is the inversion of inputs. In a similar way, Table 2.6 shows an example of obtaining an N(in)-class of 4 equivalent functions from this operation. In this case, *f1*, *f2*, *f4* and *f8* are equivalents. The last operation used is the inversion of the output. Table 2.7 illustrates this operation. Notice that the three operations can be combined. For instance, NP-classes are obtained after combining permutation and inversion of inputs. PN-classes are obtained through permutation of inputs and inversion of outputs. For NPN-classes all operations are performed.

Table 2.5: Two P-class equivalent functions.

ABC	***f2=AB+C***	**BCA**	***f4=A+BC***
000	0	000	0
001	1	001	1
010	0	010	0
011	1	011	1
100	0	100	0
101	1	101	1
110	1	110	1
111	1	111	1

Table 2.6: Four N(in)-class equivalent functions.

AB	***f1***	**A!B**	***f2***	**!AB**	***f4***	**!A!B**	***f8***
00	1	01	0	10	0	11	0
01	0	00	1	11	0	10	0
10	0	11	0	00	1	01	0
11	0	10	0	01	0	00	1

Table 2.7: Two equivalent functions after output inversion.

AB	***f9***	***f6***
00	1	0
01	0	1
10	0	1
11	1	0

Another possible classification of functions is related to their polarity behavior. **Positive unate** function is the one that presents a positive (0→1) transition in its output when a positive input variation occurs in one (or more) of its inputs. The *AND* function (*f=a*b*) is a positive unate function. **Negative unate** function, in turn, is the one that presents a negative transition (1→0) in its output when a positive input transition occurs in one (or more) of its inputs. The *NAND* function (*f=!a+!b*) is a negative unate function. **Binate function** may present both positive and negative behavior in its output when a positive (or negative) transitions are applied in one (or more) of its inputs, depending on the values of the other inputs. The *XOR* function (*f=!a*b+a*!b*) is a binate function. Notice that, unate or binate behavior in a given logic function is always related to one of its inputs; for instance the function *f=!a*b+a*!c* is binate on variable *'a'*, positive unate on variable *'b'*, negative unate on variable *'c'* and does not depend on variable *'d'*. When all inputs of a logic function have monotonic increasing behavior, then it is said that the function is positive unate in all variables. The same occurs for the monotonic decreasing behavior, which determines that the function is negative unate in all variables. *AND* and *OR* logic functions are positive unate in all input variables. On the other hand, *NAND* and *NOR* ones are negative unate in all input variables. *XOR* function is an example of binate function in all variables.

Binary Decision Diagram (**BDD**) is a data structure that can be used to represent a Boolean function. The function can be represented as a rooted, directed, acyclic graph, which consists of decision nodes and two terminal nodes called *0-terminal* and *1-terminal*. Each decision node is labeled by a Boolean variable and has two child nodes called *child-0* and *child-1*. The edge from a node to a *child-0* represents an assignment of the variable to *zero*. The edge from a node to a *child-1* represents an assignment of the variable to *one* (LEE, 1959).

Figure 2.2 illustrates a BDD of 3-input *AND* function. In this example, the function *f* is *'1'* only if *X1=1*, *X2=1* and *X3=1*. In case a variable is equal to *'0'*, the function *f* presents the value *'0'* at the output. Notice that, the nodes in a BDD are sequentially evaluated until arriving in a terminal node.

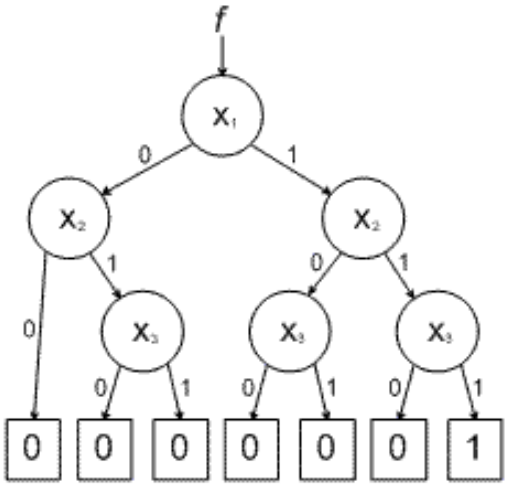
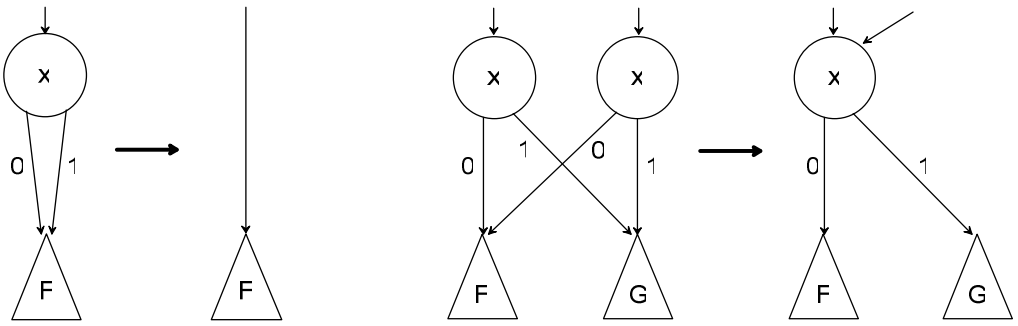


Figure 2.2: BDD of 3-input *AND* function.

The basic idea from which the data structure was created is the Shannon's decomposition. A switching function is split into two sub-functions, knows as **cofactors**, by assigning one variable. If such a sub-function is considered as sub-tree, it can be represented by a binary decision tree. BDDs are considered the state-of-the-art structure for logic synthesis because they can be efficiently used as compact and suitable representation of logic functions (EBENDT, 2005).

In Bryant (1986) a special class of BDDs is proposed. This class is known as **Reduce and Ordered BDD** (ROBDD). A ROBDD presents a fixed **variable ordering** and redundancy removal of BDD edges. The fixed variable ordering guarantees that a variable is evaluated just once along the BDD paths. The reduction of a BDD is based on two rules. The first one consists of removing BDD nodes that have their two edges connected to the same node. The second consists of sharing isomorphic nodes in the structure. Figure 2.3 illustrates these two rules.



(a) (b)

Figure 2.3: BDD reduction: (a) eliminating nodes whose two children are isomorphic and (b) merging isomorphic sub-graphs.

Due to the fixed variable ordering in ROBDDs, the canonical form concept becomes noticeable. As presented before, the canonical concept is the capability of representing a logic function in a unique form. That is, equivalent functions are represented for isomorphic structures. Notice that, in ROBDDs, the canonical concept is valid only for a given fixed variable ordering. It means that two ROBDDs representing a function *f* with variable ordering *o1* and *o2* are guaranteed to be canonical if and only if *o1* = *o2*.

Another important issue of using BDDs to represent logic functions is related to the variable ordering. The size of the BDD is determined by the function being represented and the chosen ordering of the variables. For some functions, the size of a BDD may vary between a linear to an exponential range depending upon the ordering of the variables. As presented in Drechsler (1998) and Bollig (1996) the problem of finding the best variable ordering is *NP-hard*. However, there exist efficient heuristics to deal with the problem and to obtain acceptable orders in a reasonable CPU execution time (EBENDT, 2005). Figure 2.4 shows two BDD representing the same logic function, but with different variable orderings.

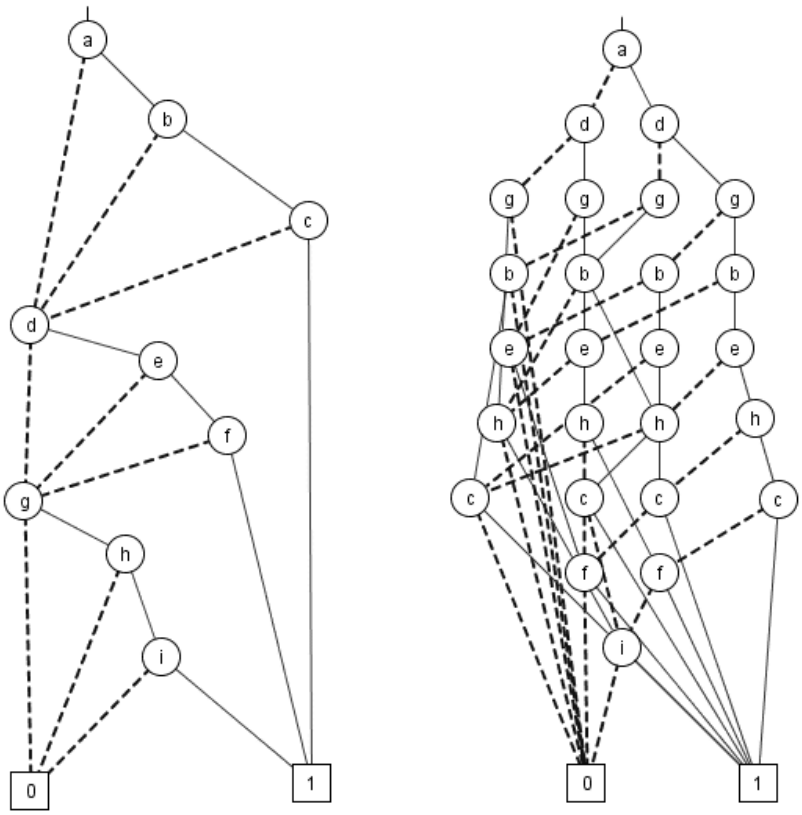


Figure 2.4: Different variable ordering ROBDDs representing a same logic function.

Examples of academic BDD packages used to manipulate Boolean functions are the CUDD (*Colorado University Decision Diagram*) developed in University of

Colorado (CUDD, 2008), and the BuDDy developed in Information Technology University of Copenhagen (BUDDY, 2008).

2.2 A Brief History of Switching Network

Switch theory is an old discipline. Back in the 30´s, when Claude E. Shannon started his work, the main logic elements were electromechanical, for instance, switches and relays. Vacuum tubes, diodes and transistors were used to make logic elements. In Shannon (1938) an analysis about relay networks and switching circuit implementation is presented. In Shannon (1953a) an investigation about how many contacts are necessary and sufficient to simultaneously realize all 16 switching functions of two variables was made. In Shannon (1853b) a machine built using selector switches and relays was conceived for helping the design of circuits composed of logic elements. In those days, the logic elements were very expensive. Also, networks to be realized were relatively small, allowing manual logic design procedures. In this context, during the 50´s (MOORE, 1958) and in the 60´s (HARRISSON, 1965), catalogs of minimum switch implementations were produced for the set of 4-input functions. Notice that, since old researches were done using relays, only the total number of switches was considered, without further investigation on how the arrangements of switches affect other characteristics of the circuit, like maximum number of devices in series and parallel. Recently, a method to determine the exact lower bound for the number of switches in series to implement a combinational logic cell was proposed in Schneider (2007). This opened the way for the generation of efficient networks having minimum length transistor chains. In the pioneer catalogs of Moore (1958) and Harrisson (1965), the lengths of transistor chains was not taken into account. Additionally, Moore and Harrisson proved that for most Boolean functions, the minimum implementation was not a series/parallel implementation. However, most of the library-free approaches are restricted to series-parallel implementations (BERKELAAR, 1988; REIS, 1998; CORREIA, 2004). Some exceptions are (JIANG, 2001) and (MARQUES, 2007). Jiang mixes pass-transistors with series-parallel implementations. Marques uses the lower bound from Schneider (2007) combined with the method presented here for the automatic generation of transistor networks with minimum chains in order to minimize the depth of a circuit in terms of transistor count. The work proposed here concentrate at the cell level, and investigates more efficient area and delay methods to optimize transistor networks taking into account the length of chains and the overall transistor counts.

2.3 Logic Switches

Several different methods have been proposed for implementing switch networks. The resulting networks may present different properties, which are not described in a comprehensive way in the literature.

The basic element to implement networks is the **switch**. This element can be called as **direct switch**, when it conducts by applying a *'1'* logic value in its control terminal, or **complementary switch**, when it conducts by applying a *'0'* logic value in its control terminal. By composing these elements, it is possible to build arrangements, known as **logic networks**, to allow the interconnection between two different terminals according to a given logic function that this network represents.

Depending of the technology used, these switches can be implemented as physical devices. In the currently CMOS technology, they are represented by the **NMOS** transistor (direct switch) and the **PMOS** transistor (complementary switch). Figure 2.5 illustrates the symbolic notation of these elements, and Figure 2.6 presents some logic networks representing arbitrary logic function.

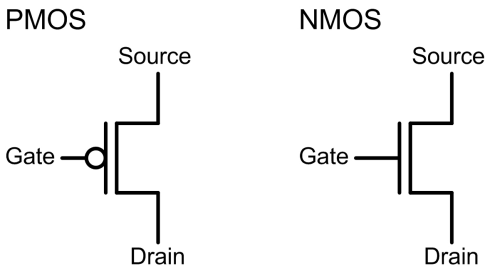
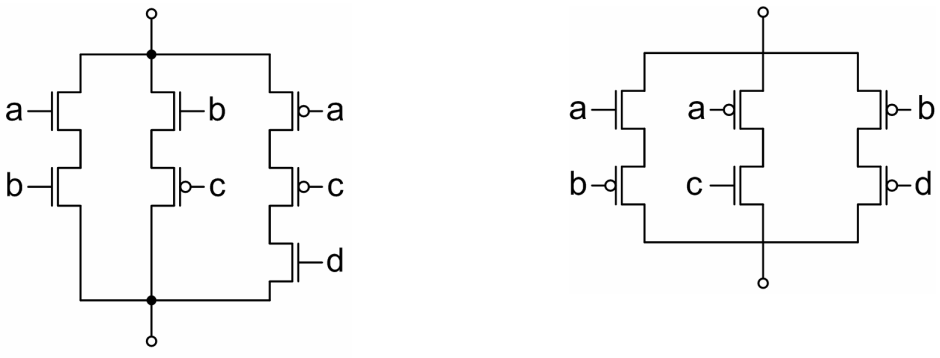


Figure 2.5: Symbolic notation for PMOS and NMOS transistors.



f = a*b + b*!c + !a*!c*d

(a)

f = a*!b + !a*c + !b*!d

(b)

Figure 2.6: Two logic networks representing arbitrary logic functions.

When looking at a single two terminal network, it may present the following properties:

- **Planar** – Networks corresponding to a planar graph (HARARY, 1994). This king of graph can be drawn in the plane without crossing lines. In the case of networks, it is additionally required that the terminals be externally connected without crossing any lines. Planar networks have a dual graph, which has the interesting property of being the logically complementary. Figure 2.7a illustrates a planar network, while Figure 2.7b illustrates a non-planar network.
- **Series-parallel** – When all switches in the network are connected in series or in parallel recursively. A network is series-parallel if and only if there is no embedded network having a Wheatstone bridge configuration (DUFFIN, 1965). All series-parallel networks are planar. This king of network is exemplified in Figure 2.8a.
- **Bridge network** – A network with an embedded network containing the Wheatstone bridge configuration. A bridge network may or may not be planar. A bridge network is never a series-parallel network. Figure 2.8b presents a bridge network.

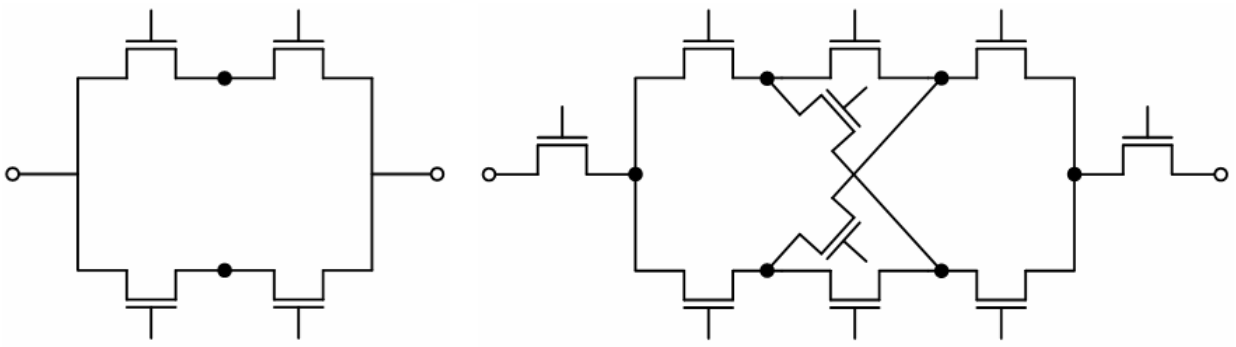
Also, some lemmas can be derived from these properties:

Lemma 1: all series-parallel networks are planar.

Lemma 2: all planar networks have a dual graph (from which a logically complementary network can be derived).

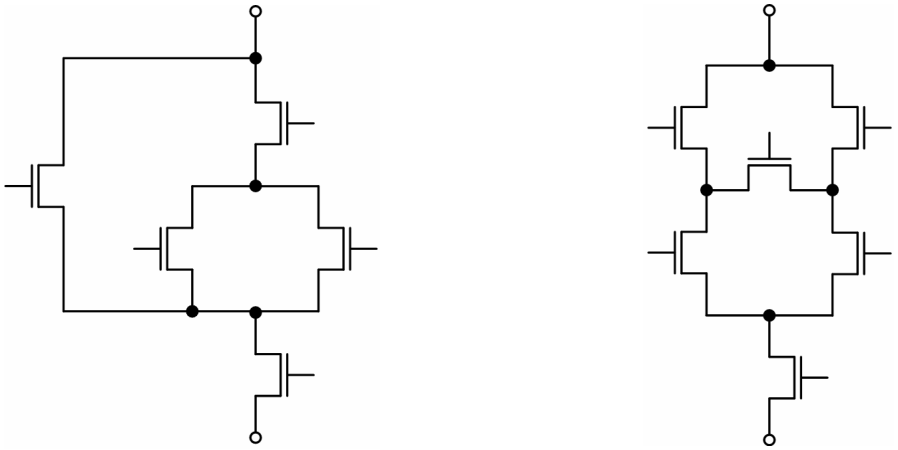
Lemma 3: all-non planar networks are bridge networks.

Lemma 4: bridge networks may or may not be planar.



(a) (b)

Figure 2.7: (a) Planar network, (b) non-planar network.



(a) (b)

Figure 2.8: (a) Series-parallel network, (b) bridge network.

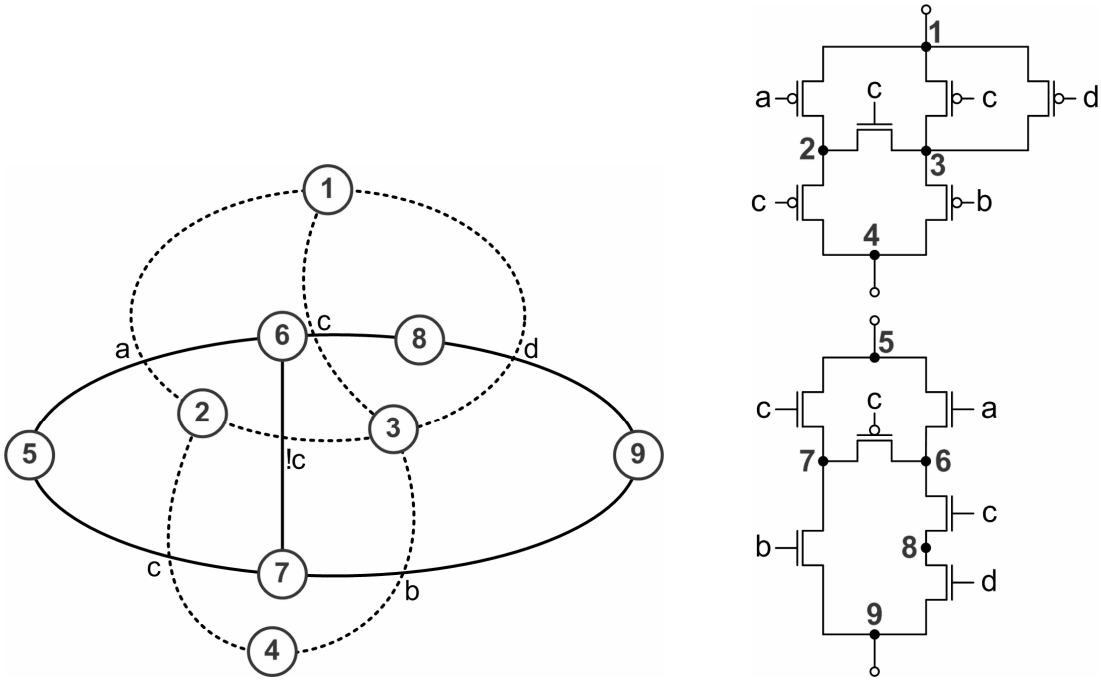
When thinking about networks composed of two planes and about complementary properties, they can be basically classified as logically and/or topologically complementary.

A network is said to be **logically complementary** when there is one and only one of the networks conducting for every input vector condition. A **topologically complementary** network is the one that presents dual planes. Figure 2.9 exemplifies this idea. The usual method of construction of the dual is the following:

1. In a given planar graph, place a point in every region of the graph. In Figure 2.9a this points are labeled as 1, 2, 3 and 4.
2. Draw all lines connecting these points through one branch of the graph. It is illustrated by the dotted lines in Figure 2.9a.

Notice that the external points, which are not inside to any internal face of the graph, correspond to the terminals. It is done for the engineering purpose. Pay attention, in graph theory, it is not necessary to set two external points to build the dual graph (HARRISSON, 1965).

It is important to keep in mind that dual networks are implemented through dual graphs. These networks are logically complementary, but they are not derived from complementary graphs. Complementary graphs are a totally different concept, which do not lead to generation of logically complementary networks.



(a) (b)

Figure 2.9: (b) Dual networks obtained through (a) dual graphs.

In the example presented in Figure 2.9b, the dual networks are bridge networks. But the same principle can be used to generate series-parallel networks, if the original graph is a series-parallel implementation. Another important point is related to the planar characteristic. If such graph is not planar, then it is not possible to derive the dual graph from it (HARARY, 1994). In this case, algorithms for graph planarization could be applied.

Branch-based is a logic network where the transistor arrangements are composed only by branches. It presents purely series-transistors connections to attach two terminal nodes (PIGUET, 1984; PIGUET, 1994; PIGUET, 1995; NÈVE, 2001). The main advantage of transistor branches is the absence of interconnections among branches, which is a positive characteristic in terms of physical design representation point of view. The construction of branch-based networks is rather simple. It takes a sum-of-products and translates each product into an *AND*-stack in the network. Figure 2.10 presents an example of a branch-based network.

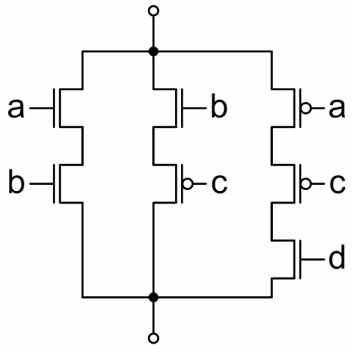
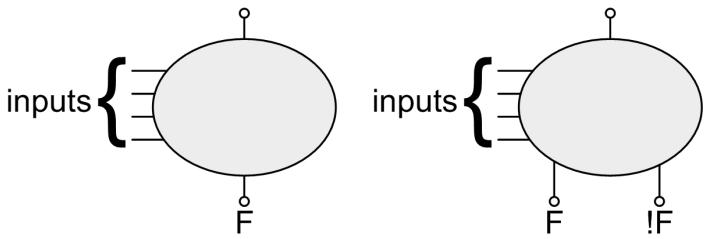


Figure 2.10: Branch-based network.

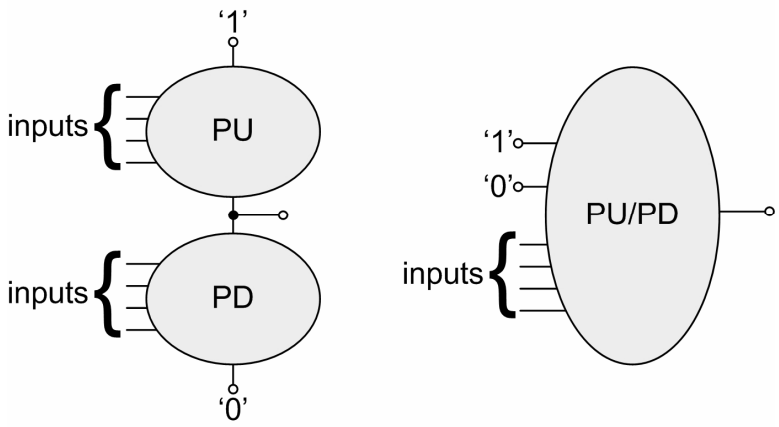
Additionally, logic networks can be also classified as **single-rail** or **dual-rail**. Single-rail networks provide the connection between two nodes. Dual-rail networks are capable of attaching one node to other two terminals, which very frequently are one for the direct polarity signal and one for the inverted polarity signal. Also, in dual-rail structures, a codification using the direct and inverted signal is done in order to guarantee the right signal propagation along circuit paths. Dual-rail logic is commonly used to build asynchronous circuits. Figure 2.11 illustrates the concept of a single and a dual-rail network.



(a) (b)

Figure 2.11: (a) Single-rail network, (b) dual-rail network.

Basically, logic network can be constructed with their logic planes in a shared structure or not. In figure 2.12a the logical network is composed of two **disjoint planes**, where the **pull-up** and **pull-down** networks are implemented separately. Figure 2.12b illustrates a logic network built in a **non-disjoint plane**, where the pull-up and pull-down networks are sharing switch elements in a single plane.



(a) (b)

Figure 2.12: (a) Disjoint planes, (b) non-disjoint planes.

The pull-up plane is the one that connects the output terminal to the *'1'* logic value, while the pull-down plane connects the output to the *'0'* logic value.

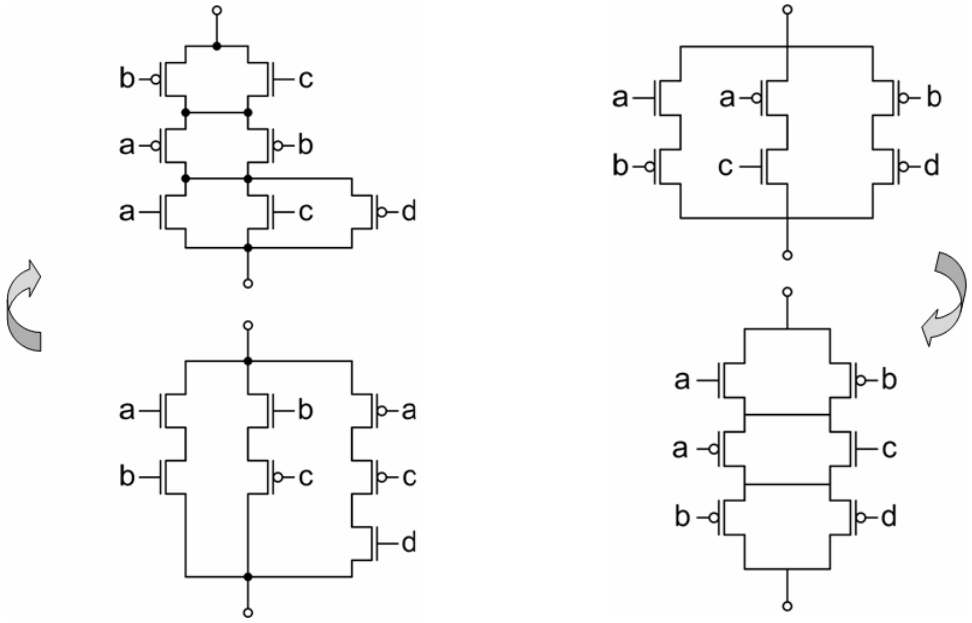
2.4 Network Generation

Two main approaches exist to synthesize switch networks. The first approach is the equation-based solution. In this approach, an equation is translated to a switch arrangement. The methods following this approach are devoted to the synthesis of series-parallel implementations, since bridge networks cannot be obtained through series-parallel association. Figure 2.13a shows a logic network obtained from the on-set equation presented in equation (2.9). Figure 2.13b illustrates a logic network obtained from the off-set equation presented in equation (2.10).

$$on\text{-}set = a*b + b*!c + !a*!c*d \quad (2.9)$$

$$off\text{-}set = a*!b + !a*c + !b*!d \quad (2.10)$$

Notice that, in both cases, it is possible to attain the topologically and logically complementary networks using the dual graph generation.



(a) (b)

Figure 2.13: (a) Network derived from the on-set and its dual network, (b) network derived from the off-set and its dual implementation.

Also it is possible to obtain the logically complementary network directly using the on-set equation to implement a given logic plane and using the off-set equation to generate the other. In this case, the obtained networks are not topologically complementary. Figure 2.14 illustrates this idea, showing the networks achieved from equation (2.9) and (2.10).

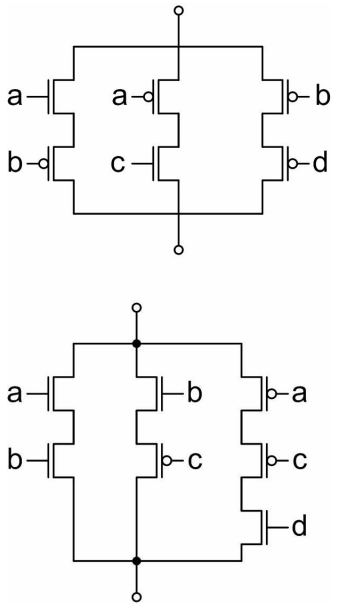


Figure 2.14: Logically complementary networks obtained from the on-set and the off-set equations.

The second approach is a graph-based solution. In this approach a graph that represents the function is created (as a BDD, for instance), optimized and then a switch network is derived from this graph. This kind of approach is interesting as it can be used to derive both series-parallel as well as non series-parallel (bridge) implementations (ROSA, 2006).

The basic action when deriving a switch network from a BDD is to associate a controlled switch to each arc of a BDD node. This concept is illustrated in Figure 2.15, which shows a BDD node and four possible ways to associate switches: transmition gates, NMOS transistors only, PMOS transistors only, and mixed PMOS/NMOS transistors (POLI, 2003).

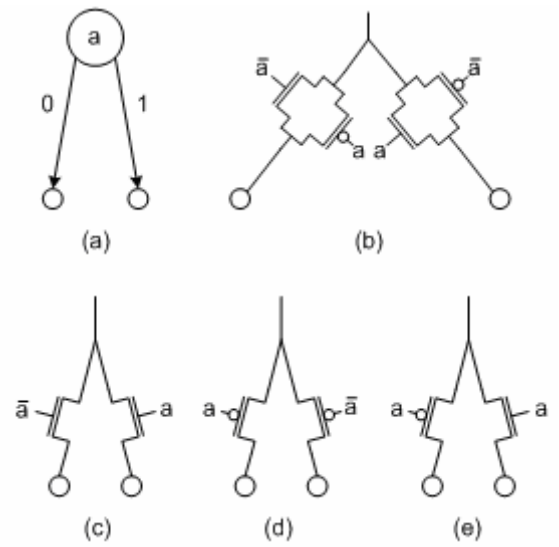


Figure 2.15: BDD node and associated switches.

When a non-disjoint transistor network is built with a pair of PMOS and NMOS transistors associated to BDD edges, there is the possibility to derive disjoint networks from it. The procedure is straightforward, as it is illustrated in Figure 2.16. Notice that in the first case, Figure 2.16a, the network in a non-disjoint and a dual-rail implementation. On the other hand, Figure 2.16b and 2.16c are disjoint and single-rail implementations.

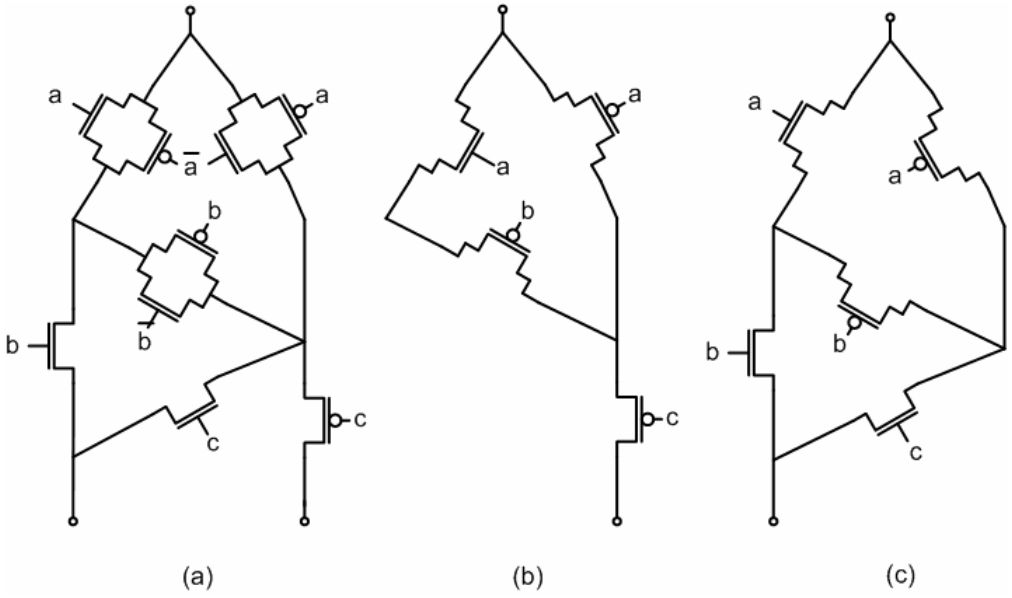


Figure 2.16: Networks derived from a BDD.

As an effect of using disjoint planes, the number of switches into the logic networks remains the same, but the number of nodes increases. Another important point is that, as the number of nodes increases while the number of elements remains the same, the number of connections to be performed among elements is reduced. This effect is visible in Figure 2.16.

The most recent work regarding switch network synthesis was developed by Kagaris (2007). In this work the authors proposed a methodology to achieve bridge networks in order to optimize the circuit in terms of transistor count. A preliminary version of it appears in (KAGARIS, 2006). The switch network is built explicitly by computing the most economical placement for the next product term of the function in the currently constructed transistor network. The most economical placement is chosen each time among several alternatives, one of which is bridging.

The basic idea of the algorithm is, from a SOP expression, to construct edges in a graph that correspond to transistors in a switch network. These edges are paths in the network and they are positioned and/or replaced in order to represent the logic function in the input SOP expression. Figure 2.17 exemplifies this procedure for a given set of terms.

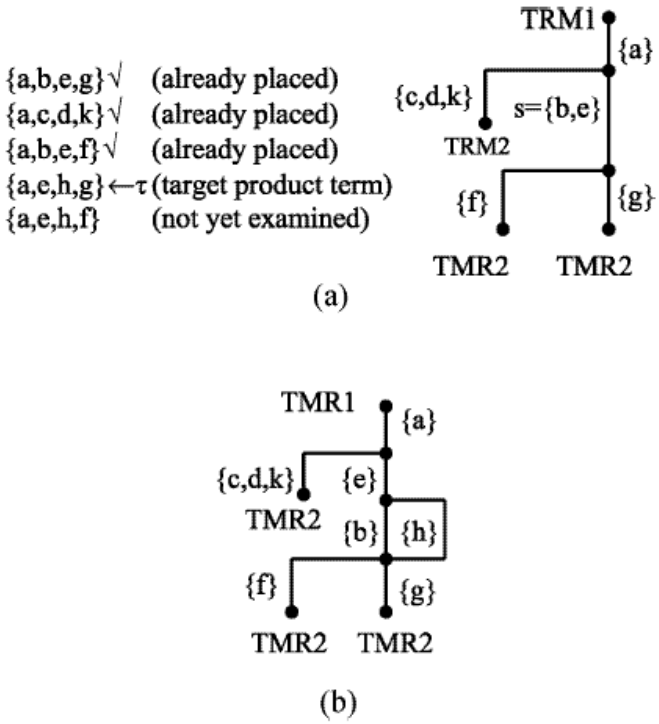


Figure 2.17: Bridge network implementation (KAGARIS, 2007).

Observe that this approach is able to generate complex gates using bridges arrangements. Nevertheless, depending of the input logic function only series-parallel networks may be achieved. Another important point is related to the logically complementary plain. The method presented in the work must be applied separated for the on-set and off-set SOPs. Thus, the two logic plains are generated in a separated way, not necessarily leading to topologically complementary solutions.

2.5 Network Optimization

One important step for the first approach is the minimization of the logic expressions. There are several methods to find the best expression descriptions. The basic idea is to find the expression with minimal number of literals. Thus, this description can be directly converted into a switch network that will present a one-to-one correspondence in numbers of literals and switch elements.

Karnaugh maps (KARNAUGH, 1953) and **Quine-McCluskey** (QUINE, 1955; MCCLUSKEY, 1956) are the main exhaustive search techniques for **two-level minimization**. Although they are typically not practical algorithms, they are easy to use and simple to understand. The **Espresso** algorithm (MCGEER, 1993) is a heuristic method for two-level minimization that is computationally less expensive and presents good results. An example of two-level minimization can be seen in the Figure 2.18. It

shows the Karnaugh map for the Boolean function *f*. The minimal cover in terms of literals for the on-set is composed by four cubes. It can be represented through the equation (2.11). Equation (2.12) shows the minimal cover in terms of literals for the off-set of the function *f*. Another possibility to the minimal cover in terms of literal is to perform a minimum cube literal cost cover, as presented in Section 2.1.

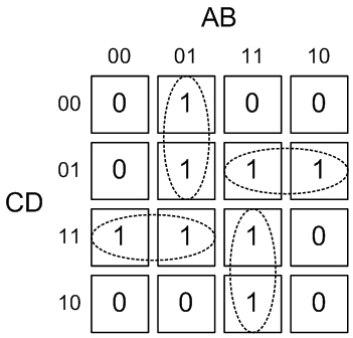


Figure 2.18: Karnaugh map for the Boolean function f.

$$on\text{-}set(f) = !a*c*d + !a*b*!c + a*!c*d + a*b*c \qquad (2.11)$$

$$off\text{-}set(f) = !a*c*d + !a*b*!c + a*!c*d + a*b*c \qquad (2.12)$$

Both equations (2.11) and (2.12) can also be represented as **factorized forms**. According to Brayton (1987), a factorized form can be defined as a representation of a logic function that is either a single literal or a sum or product of factorized forms. It is very similar to a parenthesized algebraic expression. This parenthesized representation seems to be the most appropriate representation for use in **multilevel logic** synthesis. As an example, consider the representations in the Figure 2.19. The parenthesized expression can be seen as a **logical operator tree**. Any representation with more than two levels is called a **multilevel representation**. In this example, the logical operator tree has depth four.

Some methods for obtaining different factorized forms for a given logic function are available in the literature. These **factorization methods** range from purely **algebraic** ones, which are quite fast, to so-called **Boolean** ones, which are slower but are able to give better results. Since obtaining an optimal factorization for an arbitrary Boolean function is an NP-hard problem, all practical algorithms for factoring are heuristic and provide a correct, logically equivalent formula, but not necessarily a minimal solution.

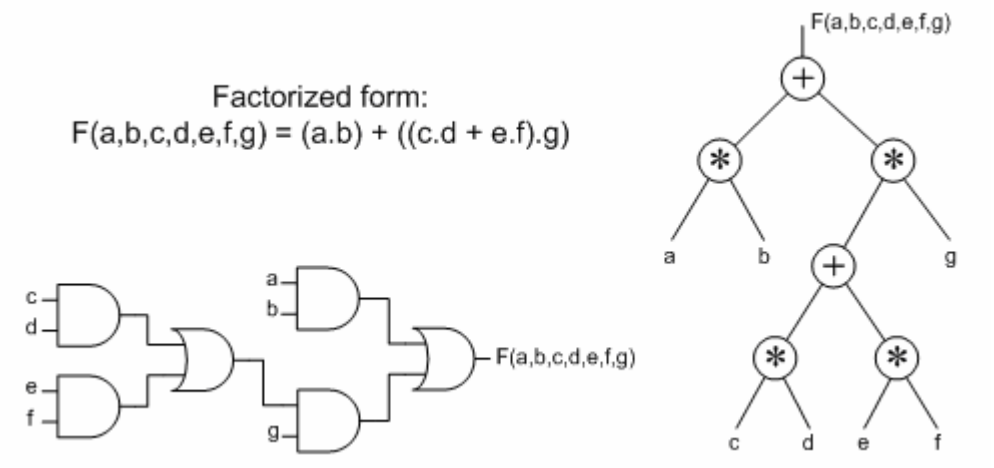


Figure 2.19: Multilevel representations.

Accorting to Mintz (2005), factorization is the procedure of deriving a factorized form from a SOP form of a function. For example, if $f = a*e+a*d+b*c$ then one possible factorization of f is $a*(e+d)+b*c$. In most logic synthesis systems Boolean functions are internally stored in the SOP form (SENTOVICH, 1992; KARMA, 2008). However, the number of elements in a switch network is more accurately represented by the number of literals in the factorized form of the network. This means that an efficient factorization method is required in order to minimize a switch network. An exact method for computing the best factorized form of a Boolean function was presented by Lawler (1964). Also, heuristics methods were proposed to provide a correct, logically equivalent form, but not necessarily a minimal length solution in tolerable computing time (BRAYTON, 1987; MINTZ, 2005).

The factorization does not influence the number of switches in series, if only algebraic operations are applied. Boolean factorization can change the number of switches in series. Both kinds of factorization (Boolean and Algebraic) affect the number of parallel branches in a network. As a result, factorization can reduce the number of switches in series in the dual of a series-parallel network. Consider the example given by the following on-set and off-set equations:

$$on\text{-}set = c*f + c*b*e + f*b*e + b*a*d + c*e*a*d + f*b*a*d + f*e*a*d \quad (2.13)$$

$$off\text{-}set = !c*!f + !c*!b*!e + !c*!b*!a + !c*!b*!d + !c*!e*!a + !c*!e*!d + !f*!b*!e + !f*!b*!a + !f*!b*!d + !f*!e*!a + !f*!e*!d \quad (2.14)$$

Equation (2.13) has four literals in the smallest cube. Equation (2.14) has three literals in the smallest cube. This way, the switch network for equations (2.13) and (2.14) is either a 3-4 (PU-PD) implementation or a 4-3 (PU-PD) implementation, depending on polarity assignment. Without factorization, the topologically complementary solution from equation (2.13) would be 4-7 (PU-PD) implementations,

while the topologically complementary solution from equation (2.14) would be a 3-11 (PU-PD) implementations. The equation (2.14) can be factorized into equation (2.15). Equation (2.15) can be used to implement a topologically complementary switch network that respects the minimum number of elements in series.

$$off\text{-}set = (!c+!f) * (!b*!e + (!a+!d) * (!b+!e)) + (!c*!f) \quad (2.15)$$

Notice that in the example above, the use of factorization allowed to achieve a solution that respects the minimum number of switches in series. However, there are examples in which factorization can reduce the overall number of switches, but it will not be sufficient to guarantee the minimum stack elements implementation given by the lower bound introduced by Schneider (2007).

2.5.1 Factorization Through Functional Composition

To obtain minimized literal cost expressions we propose a factorization method through functional composition. The main idea of this approach is to use each literal of the original equation to compose new terms and to combine these terms to each other in order to achieve a logically equivalent factorized expression. By combining small terms (with small number of literals) to generate new ones with more literals, it is possible to achieve minimized literal cost equivalent expressions in an iterative procedure.

The method consists in the following steps:

1. The input Boolean function to be factorized is added to a BDD.
2. For the cofactors of the function:
    - Verify if each literal that composes the sub function contains the cofactor, is contained in the cofactor, or not contains neither is contained in the cofactor. If the literal contains the cofactor, then it is said to be Larger Order (LO) than the cofactor. If the literal is contained in the cofactor it is said to be Smaller Order (SO) than the cofactor. If the literal not contains neither is contained, it is said to be Not Smaller or Larger Order (NO).
    - All literals are stored in a bucket of 1-element. Only literals that appear in the function are stored, disregarding literals in other polarities. This is done to optimize the algorithm, minimizing the number of combinations in the later steps.

- If a literal is equivalent to the input function, then this literal is the factorized form. It is not need to compose any term to obtain the minimized literal cost equivalent expression.
- All elements in the bucket are combined to generate new terms. These compositions are performed according to the following rules: LO *AND* LO, LO *AND* NO, NO *AND* NO, SO *OR* SO, SO *OR* NO, NO *OR* NO.
- The resulting composed elements with *'n'* literals are stored in buckets of *n*-elements if they represent different sub functions of those combined and stored before. This is a Boolean equivalence verification, which permits to eliminate several terms that do not need to be combined with any other.
- Compositions are performed to generate all possible sub functions of *'n'* literals. This way, compositions of elements with *'n'* literals are generated until any new composition could be obtained, where *'n'* goes from 2 to the maximum number of literals in a same term.

3. All terms stored in the buckets of the cofactors are unified in a same bucket of *n*-elements, discarding the NO elements. The idea is to use only the sub functions which are contained (SO) or which contains (LO) the original input function to perform the following combinations.
4. The literal that is the root variable in the BDD is stored in the same bucket. If this variable appears in positive and negative polarity in the input function, then both literals are stored.
5. Compositions are performed for all elements in the buckets, as described before. But in this step SO *AND* LO and SO *OR* LO combinations are also performed, since a final factorized form may be a composition of this sort. They main difference in this step is that the compositions are performed until to achieve an equivalent term to the input function. When this situation occurs, the minimized literal cost equivalent expression was found.

The drawback of this method is that for a large number of literals the algorithm becomes slow. This is an exhaustive solution for finding the factorized form. It is feasible for functions with 5 inputs (no more than 10 literals). For functions with a large number of literals the execution time increases due to the possible number of combinations. Also, the larger is the number of generated terms, the larger is the memory need.

Figure 2.20 exemplifies the proposed algorithm. The BDD for the input function is illustrated in figure 2.20a. The Karnaugh Map of this function is presented in

2.20b. The cofactors and their respectively Karnaugh Maps are show in 2.20c and 2.20d. Figures 2.20e and 2.20f present the buckets obtained from the cofactors. Finally, in Figures 2.20g and 2.20h, the unified bucket and the factorized expression are depicted.

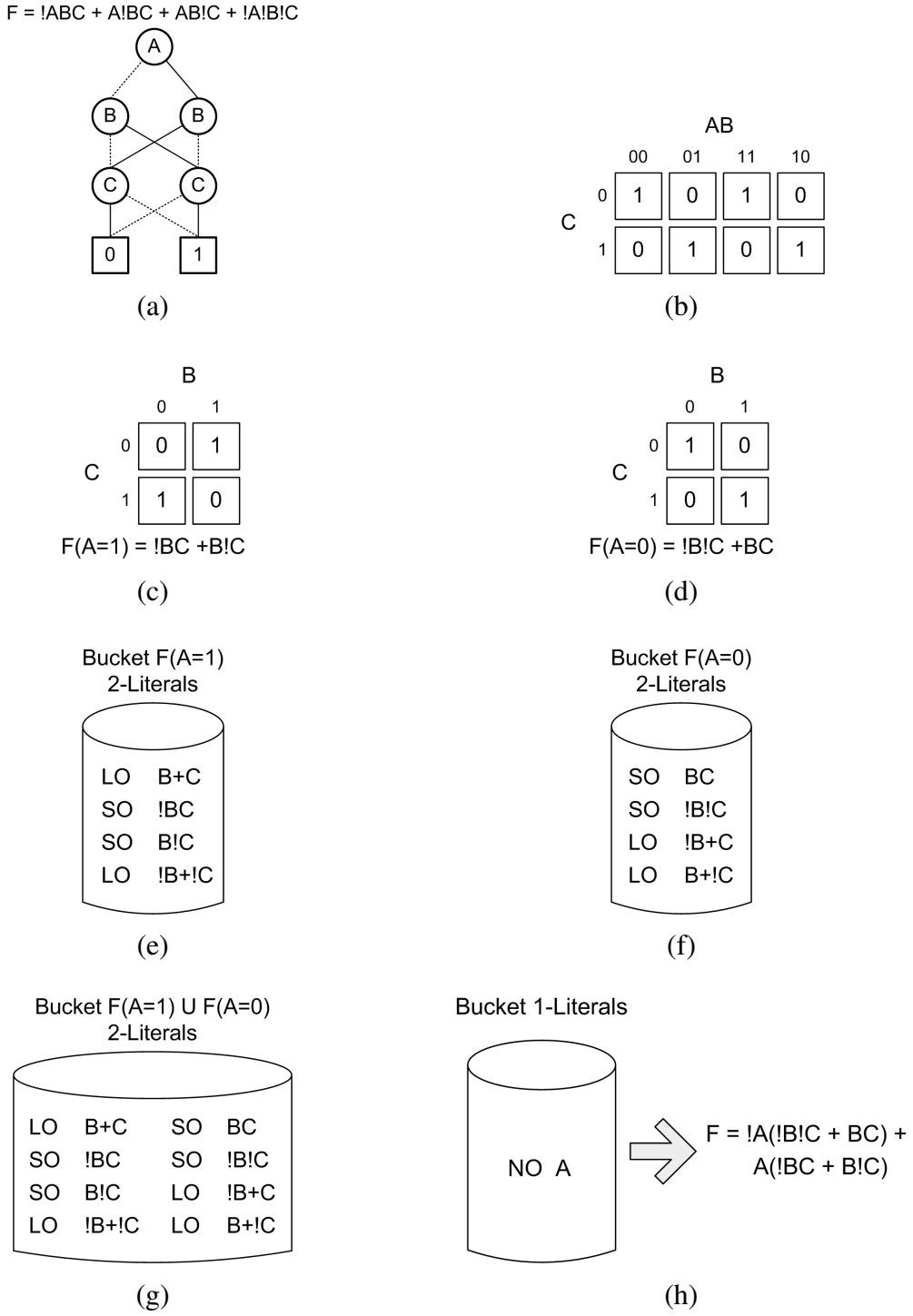


Figure 2.20: Factorization through composition

2.5.2 BDD Network Optimization through Unateness (OpBDD)

It is possible to use the unateness property of some nodes in the BDD to introduce short-circuits (wires) that do not affect the functionality of the derived network (ROSA, 2006). This approach was first used by Isaeva (1999) and Poli (2003) to reduce the transistor count.

2.5.2.1 Short-circuits and Unateness

The first concept to be understood is presented in Table 2.8, which illustrates the truth table from the function presented in Figure 2.21 separated according to the two disjoint planes. This table states the straightforward fact that when using disjoint pull-up and pull-down planes, one plane is responsible for generating the logic '1', while the other is responsible for generating the logic '0'. The logic plane that is not producing a logic value at the output produces a high impedance value *Z*. This concept will be used to prove theorem 1.

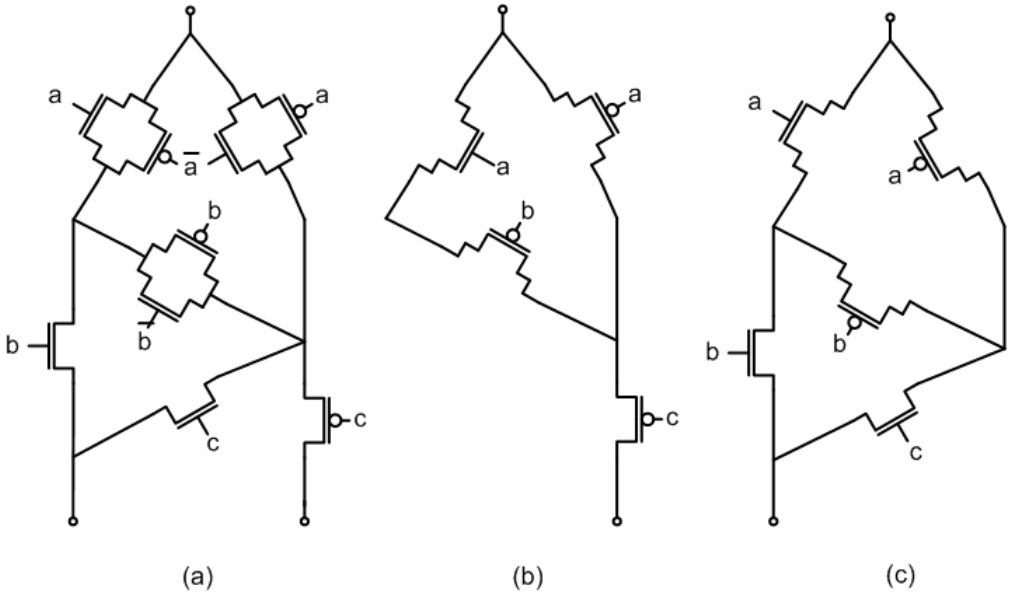


Figure 2.21: Network derived from a BDD.

Theorem 1: Given a node *'N'* in a pull-up network to be derived from a BDD, the active-1 arc of the node is a candidate to become a short-circuit, if the function represented by node *'N'* is a negative unate function with respect to the control variable *'a'*.

Proof: A node of BDD represents a Shannon decomposition such as the function represented by the node is given by the equation $f = !a*f0 + a*f1$, where $f1 =$

f(a=1) and *f0 = f(a=0)*. This equation states that *f* may be constructed from *f0* and *f1* through a pair of switches that chooses between *f0* and *f1*, as illustrated in Figure 2.15a. The truth table of this portion of the circuit states the value of *f* as a function of *'a'*, *f0* and *f1* is obtained as depicted in Table 2.9. Consider now the faulty circuit in Figure 2.22b, where the active-1 edge became a short-circuit. In order to the fault to be observable at *f*, the following conditions are necessary.

a) Variable *'a'* cannot be equal to '1', as in this case, the arc would have the functionality of a short-circuit and the fault would not be detected.

b) The value of the co-factors *f0* and *f1* must be different, in order to produce an observable fault at *f*.

c) The cofactor *f1* must be equal to logic one. If *f1=0*, it would contribute to *f* with a high impedance value *Z*, therefore the fault would not be detected, since a short connected to a high impedance value does not affect functionality. Notice that this requirement is a consequence of the creation of a disjoint pull-up plane.

As a consequence, the only combination of logic values that can detect the fault is *a=0*, *f0=0*, *f1=1*. However, if the function *f* is negative unate in variable *'a'*, by definition (of negative unateness) *f0=0* $\Rightarrow$ *f1=0*. As a consequence, if *f* is negative unate, the necessary conditions to detect the fault will never occur. This way the faulty circuit has the same functionality of the original one and the 1-active arc can become a short-circuit.

Corollary: As the considerations for proving theorem 1 are local to the node, the introduction of a short-circuit in the final network must be validated case-by-case before acceptation. Short-circuits may lead to the introduction of sneak-paths, in some cases.

Table 2.8: Truth table for function *f*, individualized by pull-up and pull-down planes.

a	**b**	**c**	**f**	**PU(f)**	**PD(f)**
0	0	0	1	1	Z
0	0	1	0	Z	0
0	1	0	1	1	Z
0	1	1	0	Z	0
1	0	0	1	1	Z
1	0	1	0	Z	0
1	1	0	0	Z	0
1	1	1	0	Z	0

Table 2.9: Truth table for function *f* and pull-up PU(*f*) as a function of *a*, *f0* and *f1*.

a	f0	f1	f	PU(f)	Fault
0	0	0	0	Z	undetected
0	0	1	0	Z	detected
0	1	0	1	1	undetected
0	1	1	1	1	undetected
1	0	0	0	Z	undetected
1	0	1	1	1	undetected
1	1	0	0	Z	undetected
1	1	1	1	Z	undetected

Similarly, this procedure of replacing transistors by short-circuits may be applied to the pull-down network. Table 2.10 illustrates the optimizations that may be done for each network plane, according to the unate characteristics.

Table 2.10: Transistor edge candidate to become a short-circuit.

	PMOS Network (*pull-up*)	**NMOS Network (*pull-down*)**
Positive Unate	Edge 0 is candidate	Edge 1 is candidate
Negative Unate	Edge 1 is candidate	Edge 0 is candidate

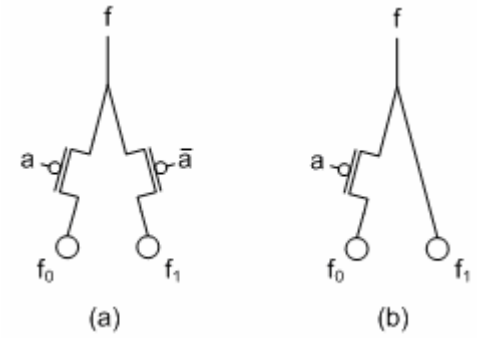


Figure 2.22: Switches controlled by variable *a* are used to choose between cofactors *f0* and *f1*.

2.5.2.2 Dominance and Open-Circuits

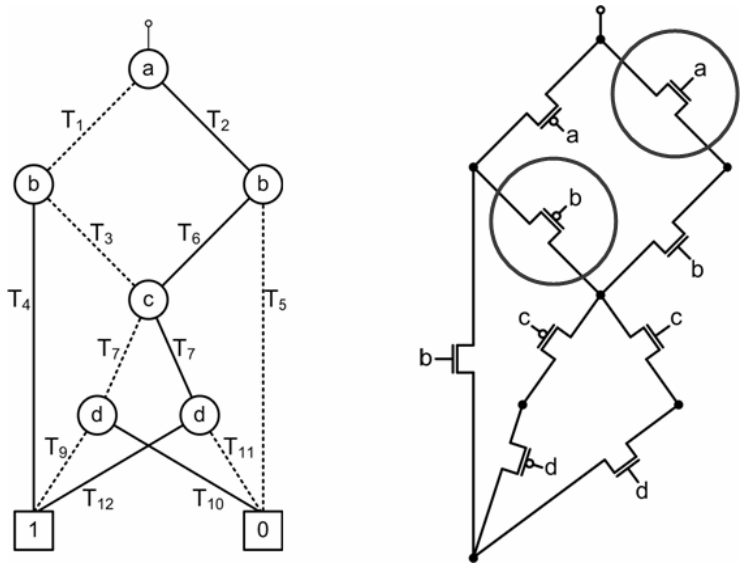
The introduction of short-circuits, as described previously, can introduce a dominance relationship between paths. A path *P1* in a network dominates a path *P2* if *P1-on* $\Rightarrow$ *P2-on*. This is the case of the pull-down network presented in Figure 2.21c. There are three paths connecting the terminals: *P1* = *a*b*, *P2* = *!a*c* and *P3* = *a*!b*c*. The path *P2* could be simplified to *P2* = *c*, by forcing the active-0 arc of variable *'a'* to become a short-circuit. After this reduction, *P2* dominates *P3*, because *P2-on* $\Rightarrow$ *P3-on*.

As a consequence, *P3* is not needed for achieving the right functionality for the circuit, and *P3* is removed by making the active-0 arc of variable *'b'* an open circuit.

2.5.3 Lower Bound BDD Network (LBBDD)

Sometimes not only optimizations through unateness are sufficient to guarantee that the generated network will respect the minimum number of transistors in series. Essentially, that occurs because some transistors, which are candidates for optimization, cannot be replaced by a short-circuit or by an open-circuit during the optimization process since this might lead to an introduction of invalid paths into the circuit. Figure 2.23 illustrates this concept, where a BDD and its derived disjoint plane are shown. The circled transistors in the switch network, Figure 2.23b, are candidates for being replaced by short-circuits. The T2 transistor can be replaced by a short-circuit as this optimization does not affect the original behavior of the logic function. However, the T3 transistor cannot be optimized because it might activate the invalid path through T4 and T6 arcs, changing the logic function behavior. To assure the correct functionality of the network, node duplication can be applied to the original BDD or network structure.

Node duplication can be performed through different ways. Basically, we can analyze the process in two separated approaches: the structural choice and the duplication strategy.



(a) (b)

Figure 2.23: BDD and derived switch network.

2.5.3.1 The Structural Choice

First of all, it is necessary to choose the appropriate structure to duplicate nodes of a given logic function. Generally, BDDs packages are implemented with ROBDDs structures. This is due to ROBDDs being a special class of BDDs that have a reduced and ordered structure to represent logic functions. Therefore, it will be required a different structure to keep the modified BDDs, if a ROBBD package is used to store and implement transistor networks. Another alternative to apply node duplication is to do it directly into the transistor network. In this method, the BDD structure that stores the logic function is not modified and the duplications are performed in the network nodes. As a consequence, all new transistors created during the duplications can be stored in the existing transistor list and no new structure is necessary to keep the modifications. This alternative is extremely efficient in terms of memory requirement and execution time as the algorithms for performing it are considerably simple.

2.5.3.2 Duplication Strategy

Another issue about node duplication is related to the choice of the BDD node which the duplication will be applied. The Figure 2.24 illustrates two different duplications performed on a switch network derived from the same BDD presented previously on Figure 2.23. In order to allow transistor T3 to become a short-circuit, two specific duplications may be applied to the network. The first option is to duplicate node *'C'* of the BDD, consequently, to duplicate T7 and T8 transistor in the network. Figure 2.24b illustrates this idea. The second option is to perform the duplication on the node *'B'*, which has the candidate transistor connected. This is shown in the Figure 2.24c, where T1 transistor is duplicated. The main reason for evaluating these strategies is to determine the number of switches that will compose the network. In this example, the two switch networks respect the minimum transistor stack. However, the second one is better as it has one transistor less than the first implementation. If considering circuits composed by a significant number of gates, the increase of unnecessary switches might become a problem in terms of area consumption.

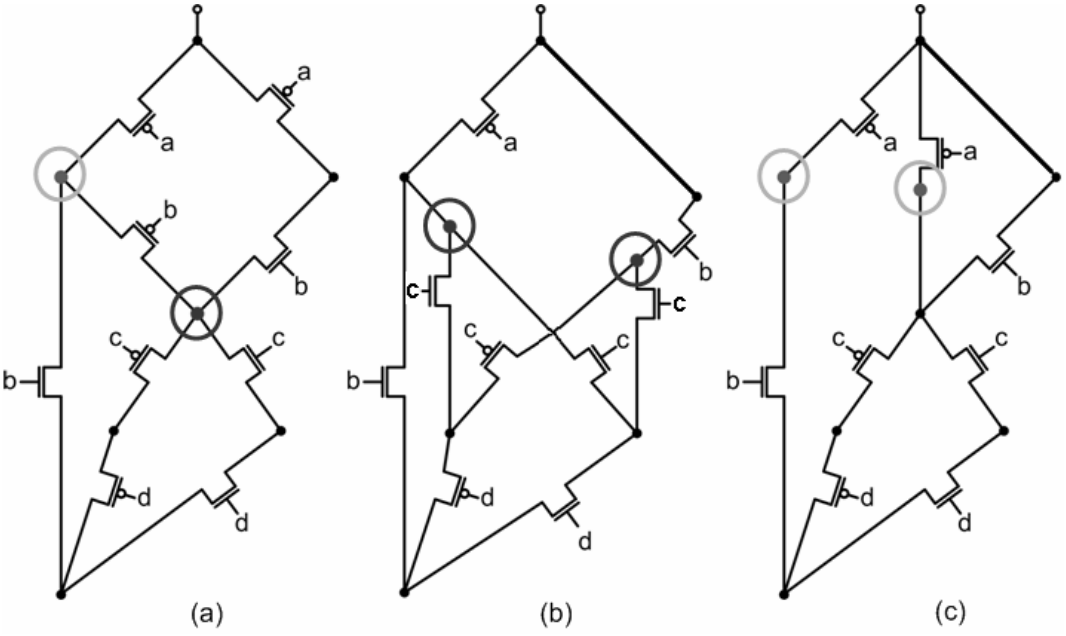


Figure 2.24: Duplication strategies for a switch network.

2.5.3.3 Implemented Methodology to Achieve Minimum Transistor Stacks

By using the observations described previously, an optimization method for generating networks with minimum transistor stacks from BDDs was implemented (ROSA, 2007). In this proposed approach, the duplications are performed directly in the transistor network to avoid the need of any additional structure to store data and also to avoid unnecessary use of memory. In addition, the two duplication strategies were implemented to guarantee a network with a minimum number of transistors. The implemented method is basically divided into two steps to achieve the lower bound of transistors in series. These steps are presented as it follows.

Step 1: In this first step a network with disjoint pull-up and pull-down planes is generated. In sequence, all the unate nodes in the BDD are identified and the potential transistor candidates are separated into a reference list to be optimized subsequently. All candidates that are separately connected to a given node are turned into short-circuits. If there is a node which has both a candidate and a non-candidate transistor connected, a duplication process is applied and the candidate is replaced by a short-circuit. Furthermore, the second strategy of node duplication is performed. Thus, two networks are generated and the best one is chosen (the one which has the lower number of transistors). This procedure is executed as many times as necessary in order to guarantee that all candidate transistors are replaced by short-circuits. After performing all duplications and replacements, a comparison between the lower bound and the number of transistor in series of the network is performed. If the network respects the minimum number of transistor in series the process is finalized, if it does not, the second step is executed.

Step 2: Generally, the execution of step 1 is sufficient to guarantee that the network length respect the minimum number of transistor chain. However, sometimes it is necessary to remove some transistors that cannot be identified as candidates through unateness. This situation is illustrated in the Figure 2.25, which the BDD and a derived switch network are shown. After executing all possible optimizations, the transistor network, Figure 2.25b, presents three transistors in series. The minimum number of switches in series for this function is two, what means at least one transistor in this network must be removed. To solve this problem the method analyzes all the paths in the network. The transistors that belong to those paths that exceed the lower bound are selected as candidates to be removed. One by one, they are replaced by short-circuits and a functional simulation is performed to verify the network reliability. If the simulation result is as expected and there is no more paths exceeding the lower bound, the process is finalized. If the result is not as expected, the previous network is restored, another transistor is removed and the network is simulated again. This process is repeated until the method acquires a network that respects the minimum transistor stacks and that corresponds to the original logic function. In this example the T6 transistor will be removed and, as a consequence, T11 and T15 will be just as well.

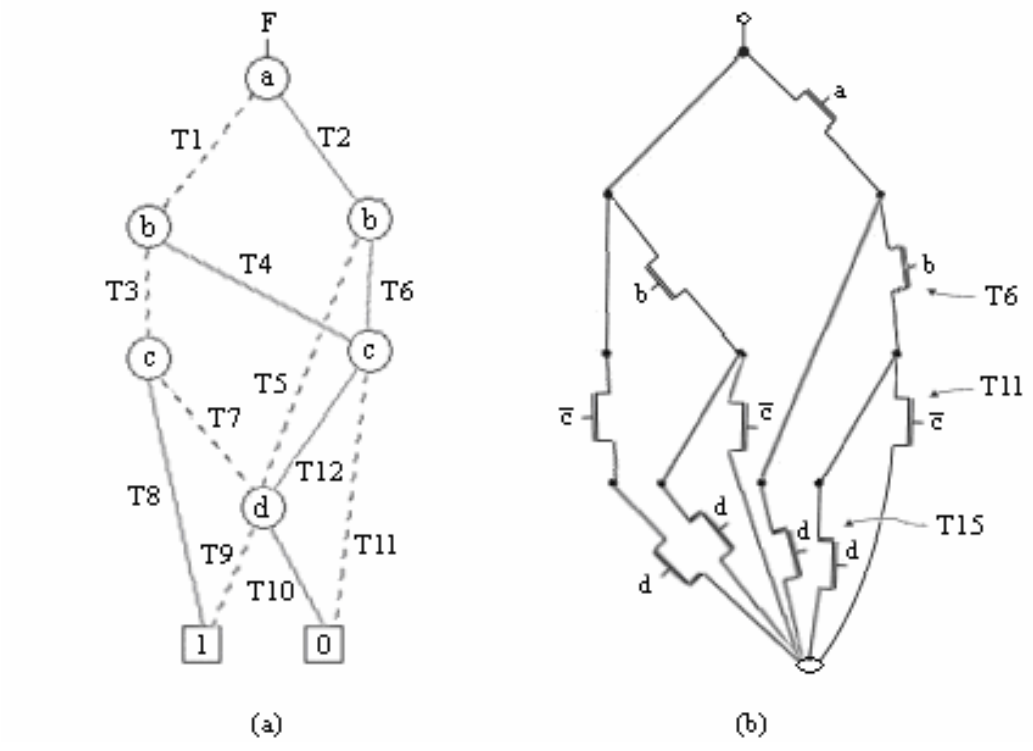


Figure 2.25: BDD and optimized switch network.

2.6 Network Ordering

Switch ordering is another approach used to reach network optimizations. The idea is to organize the internal switch arrangement in order to get a better network implementation in terms of a given cost. This concept basically may be classified as **structural ordering** or **graph-oriented ordering**.

In structural ordering the network is generated in a first step, and, in a second step, the switches are placed according to some rules to minimize a specified cost. Only elements connected in series in the network can be ordered to produce a new network. This is a heavy restriction as it depends on the initial topology of the network. For example, it is possible to favor some input signals putting the switches that control these signals close to the output. Thus, the network will present better performance for a certain input signals, since the distance to the output of the switches controlled by these signals will be minimized. The method proposed by Carlson (1992) is an example of structural ordering technique.

In graph-oriented ordering, the data structure is ordered before generating the switch network. The idea is the same that structural ordering, but in this case the switch network will be generated in a given ordering previously defined in the data structure; this way, all orderings are possible and there is no restriction imposed by the original graph. The method proposed by Cardoso (2008) is an example of graph-oriented ordering technique performed in BDD.

The BDD size may present from a linear to an exponential relation according to the number of variables present in the graph, depending of the represented logic function and the variable ordering. It was demonstrated that finding the BDD ordering that present the minimal number of nodes is a NP-hard problem (BOLLIG, 1996; DRECHSLER, 1998). The amount of possible ordering is determined by the factorial of the variables present in the logic function. Thus, it is only possible to obtain good solutions using exhaustive approaches for a small number of variables considering acceptable execution time. In the practice, heuristic methods like *sifting* (RUDELL, 1993) may deliver good results in reasonable time for larger BDDs. The *sifting* method consists to sequentially swap a variable for all BDD levels and, in a greedy strategy, to fix it in the position that the BDD presents the smaller number of nodes. In general, BDD ordering methods are based in the swap of adjacent variables.

Notice that finding the best BDD ordering means to find the BDD with minimum number of edges. If think that each edge of the BDD is translated in a transistor element, this strategy is extremely important in order to achieve more optimized networks. Figure 2.26 illustrates two BDDs representing the same logic function and the transistor networks obtained from them. Small BDDs are translated in small networks.

For this work two approaches were implemented. The first one is an exhaustive solution, were all variable ordering are tested. This approach can be applied for BDD up

to eight variables. For BDD with more than eight variables, the *sifting* algorithm was implemented.

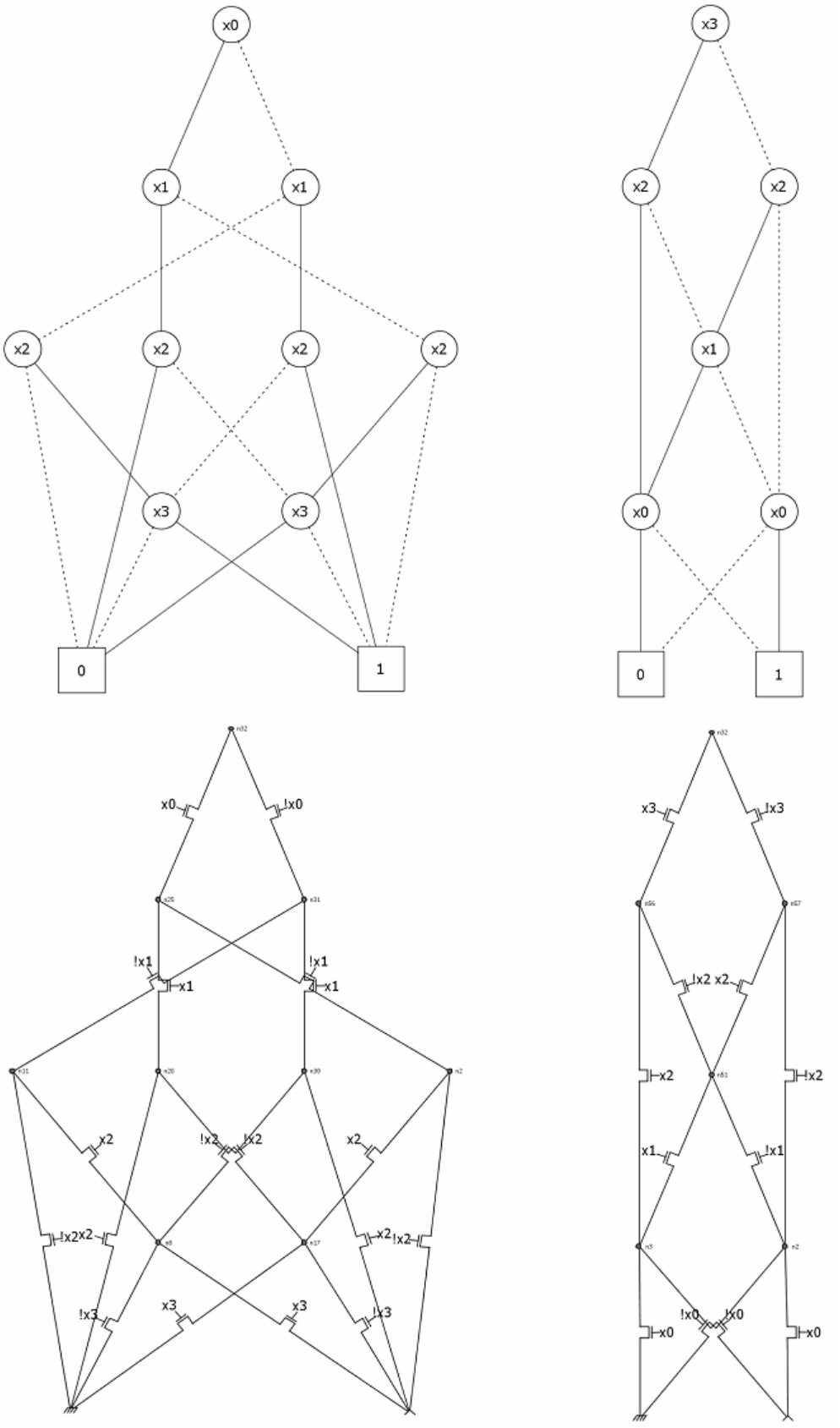


Figure 2.26: Two BDDs and their derived transistor networks.

2.7 Conclusions

This chapter presented some basic concepts and terminologies regarding logic synthesis and switch networks. A brief history about switch theory was shown, as well as networks generation and optimization. A method to factorize Boolean functions was presented in order to achieve minimum literal cost expressions, and a transistor network derived from BDD, capable of respecting the minimum number of transistors in series, was also proposed.

3 CMOS LOGIC STYLES

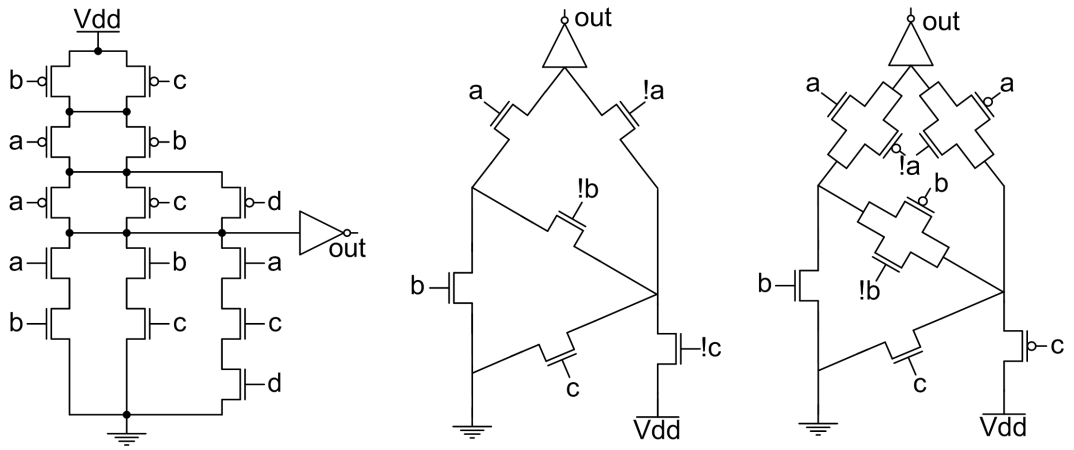
There are several works in the literature about transistor networks and CMOS logic styles. This chapter discusses this topic, presenting alternative logic styles to the traditional CMOS standard. In the sequence, a classification is done for two terminal disjoint networks. Timing, power and layout are also discussed herein.

3.1 Logic Styles

Logic styles are basically classified as being dynamic or static topologies. **Dynamic styles** rely on temporary storage of signal values on the capacitance of high-impedance circuit nodes (THORP, 2003). The implementation approach of dynamic circuits is simpler and faster but their design and operation are more prone to failure because of the increased sensitivity to noise. The most common dynamic logic styles are Domino and its variants Dual Domino, Multiple-Output Domino, NORA Domino and Zipper Domino (WESTE, 2005). On the other hand, **static styles** guarantee that, under fixed input vectors, each gate output is connected to either *Vdd* or *Vss* via a low resistance path. Also, the outputs of the gate assume at all times the value of the Boolean function implemented by the circuit, meaning the circuit does not need to be pre-charged or pre-discharged. Some of the most common static logic styles are Static CMOS, Pseudo-NMOS, DCVSL and PTL (RABAEY, 2005).

The most used logic styles used in the industry are the **complementary series/parallel CMOS** (indicated here as **CSP**) and the **pass-transistor logic** (**PTL**), both static and single-rail topologies. Accordingly to Weste (2005), the usual static CMOS has the important characteristic of low static power consumption, if compared to dynamic logic. Significant power is only drawn when the MOS transistors devices are switching between on and off states. Traditionally, logic cells have been implemented using static CMOS due to its good performance, advantageous noise immunity, and easy and widely known design methodology (LAI, 2006). On the other hand, PTL logic style is a promising alternative, since it may employs NMOS transistors only that have small capacitance, which may reduce the power dissipation while offering similar

performance as static CMOS (BERTACCO, 1997). Also, PTL presents a potential reduction of transistors count. For instance, PTL logic style is known for better implementations as compared to static CMOS in case of arithmetic circuits, such as adders and multipliers where Exclusive-ORs (XORs) dominate (RUPESH, 2004). Figure 3.1 illustrates the static CMOS (CSP) and PTL logic styles.



(a) (b) (c)

Figure 3.1: Logic styles: (a) Static CMOS, (b) PTL using only NMOS transistors, (c) PTL using transmission gates.

This thesis focuses on generating and evaluating static, single-rail and disjoint networks. The next sections will present some logic styles that follow these characteristics.

3.1.1 Complementary Series-Parallel CMOS (CSP) Network

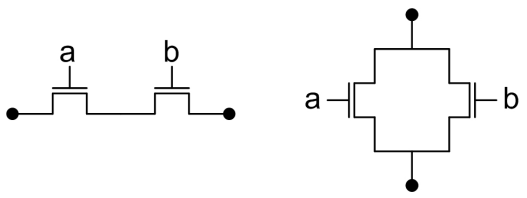
The CSP is currently the most used and well established logic style applied by the industry. This style is essentially an extension of the CMOS inverter to multiple inputs. The major advantage of the CSP structure is the low sensitivity to noise, good performance and low power consumption with almost no static power consumption for technologies with transistor channel length down to 130nm (WESTE, 2005).

CSP gate is a combination of two networks, one to build a pull-up plane and another to build a pull-down plane. Figure 3.1a shows a logic gate where all the inputs are distributed to both the pull-up and pull-down planes. As mentioned before, the goal of the pull-up plane is to provide a connection between the output and *Vdd* anytime the output of the logic gate is meant to be ‘1’, based on the input signals. In the same way, the task of the pull-down is to connect the output to *Vss* when the output of the logic gate is meant to be ‘0’. The pull-up and pull-down networks are constructed in a

mutually exclusive mode such that one and only one of the networks is conducting in steady state. In this way, once the transients have settled, a path always exists between *Vdd* and the output, realizing a high output (representing logic '1'), or, alternatively, between *Vss* and the output for a low output (representing logic '0'). This is equivalent to stating that the output node is always a low-impedance node in steady state.

While constructing CSP pull-up and pull-down networks, the following observations should be considered:

- The pull-down is constructed using NMOS devices, while PMOS transistors are used in the pull-up. The primary reason for this choice is that NMOS transistors produce "strong zeros" and PMOS devices produce "strong ones" (RABAEY, 2005).
- A set of construction rules can be derived to construct logic functions. NMOS devices connected in series correspond to an AND function, as shown in Figure 3.2a. Similarly, NMOS transistors connected in parallel represent an OR function, as illustrated in Figure 3.2b.
- Using similar arguments, construction rules for PMOS networks can be formulated. But in this case, the complementary property can be considered. This means that a parallel connection of transistors in the pull-up network corresponds to a series connection of the corresponding devices in the pull-down network, and vice versa. Therefore, to construct CSP logic, one of the networks is implemented using combinations of series and parallel devices. The other network is obtained using the duality principle by traversing the hierarchy, replacing series sub-nets with parallel sub-nets, and parallel sub-nets with series sub-nets. The complete CSP logic is constructed by combining the pull-up with the pull-down (WAGNER, 2006).



(a) Series (b) Parallel

Figure 3.2: NMOS logic rules: (a) series devices produce an AND operation, (b) parallel devices produce an OR one.

Notice that the idea of constructing a CSP logic network is the same that was present in Section 2.4. During the 90´s, some BDD-based methods were proposed to generate this kind of network (REIS, 1995; GAVRILOV, 1999; LIU, 1999). In practice,

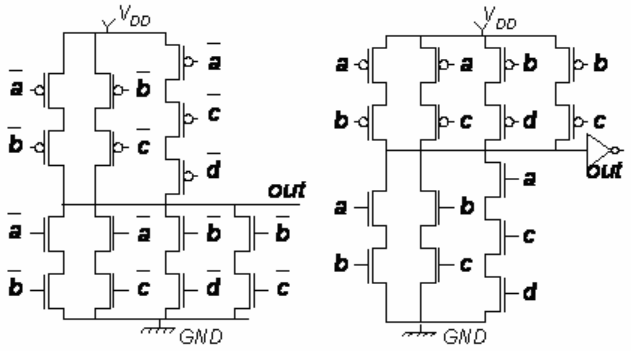
they are not useful since there is no need to have a complex and CPU costly algorithm to achieve a transistor network that can be directly derived from an equation description.

3.1.2 Gates with Minimum Transistor Chains (NCSP)

It is possible to derive transistor implementations for a given logic function, while guaranteeing minimum length transistor stack in the derived network (SCHNEIDER, 2006; ROSA, 2007). The methodology to determine the minimum possible length for the implementation of a logic function is the one presented in Section 2.1, where a minimum cube literal cost SOP is applied. After that, to obtain gates with minimum length pull-up and pull-down chains, the method generates a pull-up plane from the on-set equation and a pull-down plane from the off-set equation, as presented in Section 2.4. If it is desired, the inversion of the input logic function *f* leads to a "deMorgan" implementation that exchanges the pull-up and pull-down planes. Figure 3.3 shows the NCSP implementations for the function represented by equations (3.1) and (3.2).

$$on\text{-}set = a*b + b*c + a*c*d \quad (3.1)$$

$$off\text{-}set = !a*!b + !a*!c + !b*!d + !b*!c \quad (3.2)$$



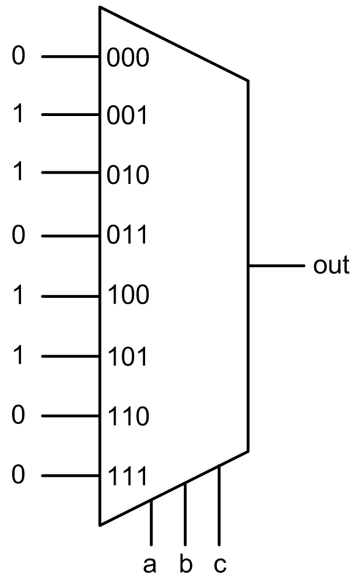
(a) 3-2 cell for direct *f* (b) 2-3 cell for inverted *f*

Figure 3.3: NCSP implementation for equations (3.1) and (3.2).

3.1.3 Mux-Based Network

A multiplexer is a combinational circuit that has 2^n binary inputs and *'n'* control inputs. Its output corresponds to the binary input selected by the control inputs. According to Ercegovac (2000), a 2^n inputs multiplexer may be used to implement any logic function with *'n'* inputs. It is possible if the input variables of the function are

used as control inputs of the multiplexer, and the output values of the function are used as binary inputs, to be selected by control inputs. Figure 3.4 shows a multiplexer implementing an arbitrary function.



a	b	c	out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 3.4: Mux_8x1 implementing a logic function.

However, it is possible to reduce the multiplexer input number if, beyond the constants '0' and '1', a variable or its complement is connected to the multiplexer data inputs. Thus, a 2^n multiplexer can implement any function of $n+1$ variables. To find the signal that should be connected to the multiplexer inputs, it is necessary to perform a simplification over the SOP of the function, choosing the variable to be used as input. The following equations exemplify this procedure:

$$out = !a*!b*e1 + !a*b*e2 + a*!b*e3 + a*b*e4 \quad (3.3)$$

$$out = !a*!b*e1 + !a*b*e2 + a*!b*1 + a*b*0 \quad (3.4)$$

$$out = !a*!b*c + !a*b*!c + a*!b \quad (3.5)$$

Figure 3.5 illustrates the multiplexer obtained for the simplification described above.

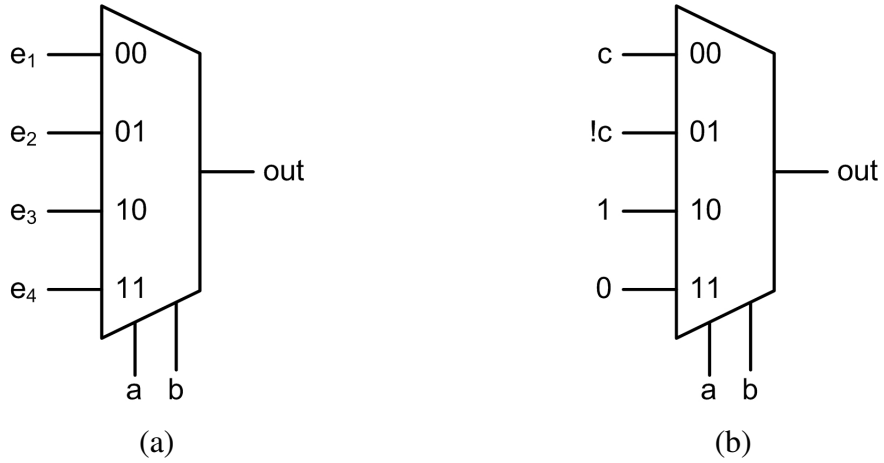


Figure 3.5: Mux_4x1: (a) generic symbol, (b) implementing function from Figure 3.4

From the electrical point of view, a multiplexer may be implemented using tri-state inverters. It is a quite simple procedure, since the generated structure is regular. To do this, it is necessary to connect the control variables of the multiplexer to the control variables of the tri-state inverters, and connect input signals of the inverters to the input signals of the multiplexer. Figure 3.6a illustrates this idea and Figure 3.6b shows a transistor network for the mux of figure 3.5.

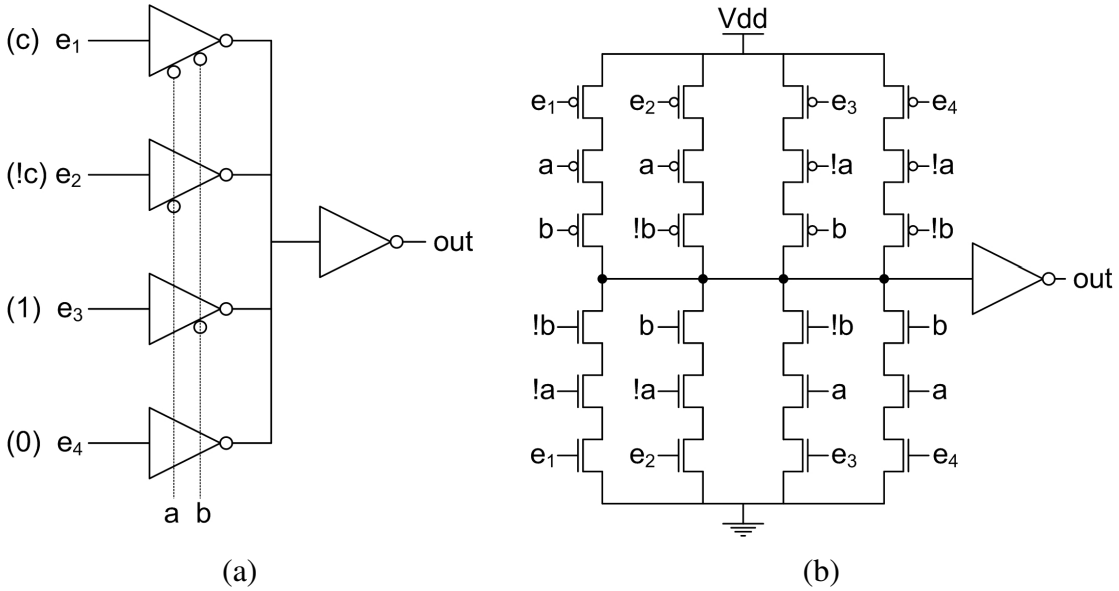


Figure 3.6: (a) Mux using tri-state inverters, (b) mux_4x1 transistor network.

Notice that, there is the possibility to optimize the mux-based network. Usually, it is done when a transistor input is permanently connected to *Vdd* or *Vss*. This situation leads to short-circuit and open-circuit transistors insertion in the network. These transistors may be removed from the network without modifying its logical

behavior. Figure 3.7 exemplifies this idea, where the total number of transistors in the network is minimized.

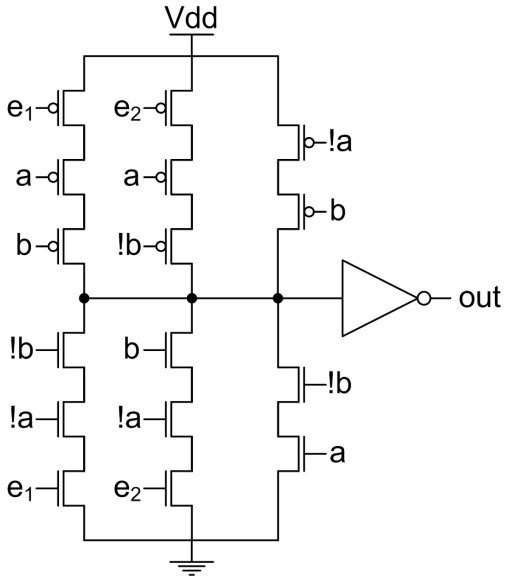


Figure 3.7: Optimized mux_4x1 transistor network.

3.1.4 BDD-based Networks

Using the switches association described in Section 2.4, and the tricks described in Sections 2.5.2 and 2.5.3, disjoint BDD networks can be implemented. To do that, the pull-up plane is built using PMOS transistors, while the pull-down plane is built using NMOS transistors. Figure 3.8 shows a BDD and the disjoint networks derived from it.

Notice that this logic style is capable of delivering bridge networks, which are not possible in the previous networks. Also, like NCSP networks, it is possible to achieve networks that respect the minimum number of transistors in series.

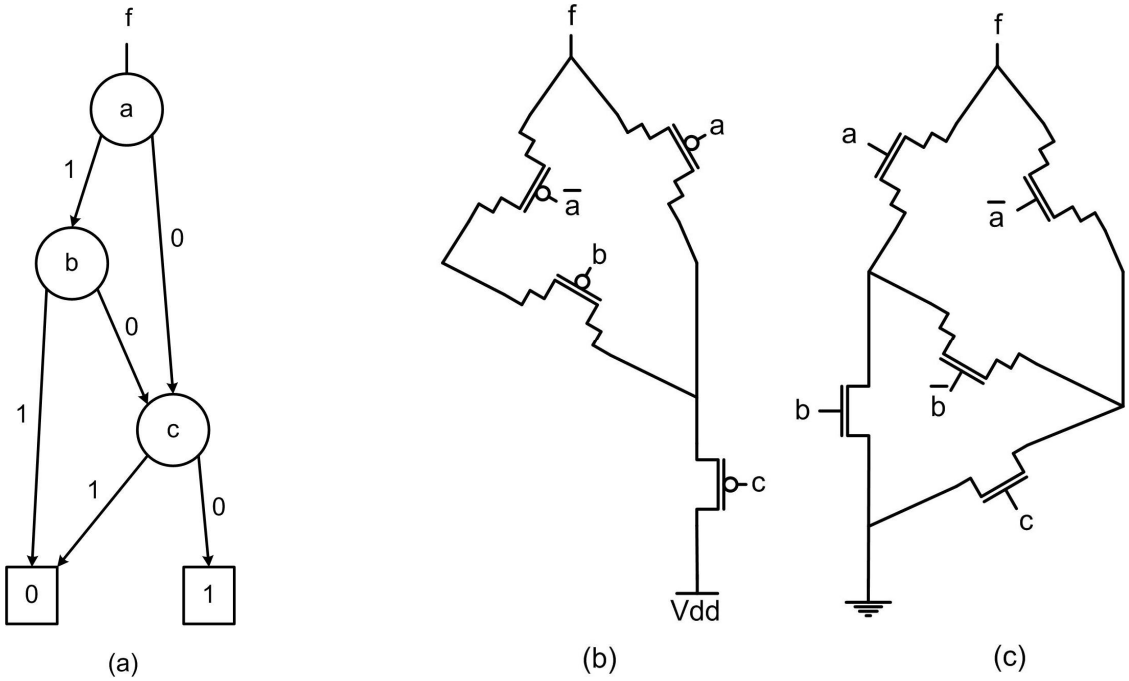


Figure 3.8: (a) BDD representation, (b) pull-up network and (c) pull-down network derived from it.

3.2 Classification of Two Terminal Disjoint Networks

When analyzing pairs of networks it is possible to identify the following properties:

- **Series-parallel complementary** – When the graphs of pull-up and pull-down networks are series-parallel and one is the dual of the other. Notice that from a graph theory point of view, dual and complementary graphs are distinct concepts.
- **Topologically complementary** – When the graphs of pull-up and pull-down networks are dual. With respect to the previous definition, the request of being series-parallel was removed.
- **Logically complementary** – When there is one and only one of the networks conducting for every input vector condition.
- **Self-dual** – When pull-up and pull-down have exactly the same topology, including the variables controlling the switches.
- **Short circuit** – When there is one input vector where both pull-up and pull-down networks conduct, such that *Vdd* and *Vss* are short circuited.

- **Tri-state** – When there is one input vector where neither pull-up nor pull-down network conducts. Consequently the output is let on high impedance state.

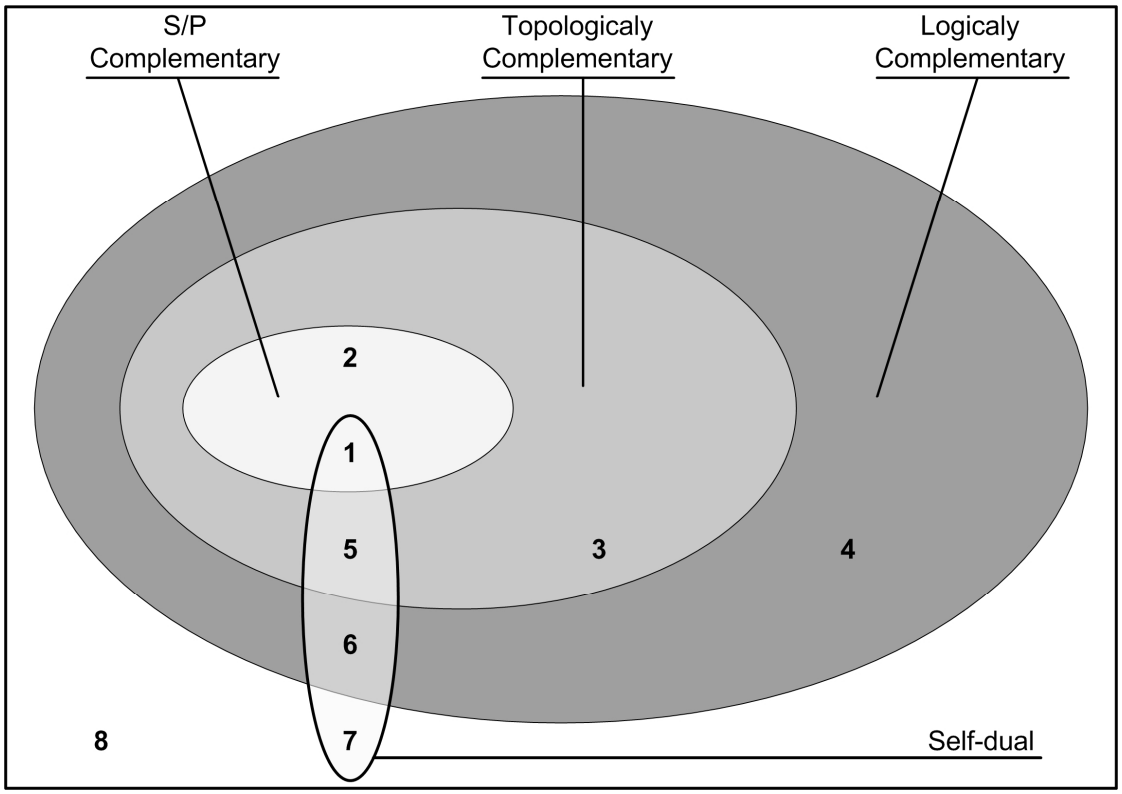


Figure 3.9: Classification of two terminal disjoint networks.

Figure 3.9 illustrates the classification described above. Notice that there is only one network of type 1. This network is the inverter, which is self-dual, series-parallel, topologically and logically complementary. The group 2 is composed of all series-parallel complementary networks. For example, NAND2, NOR2 belong to group 2. These networks are also topologically and logically complementary. Groups 3, 4, 5, 6, 7 and 8 do not present series-parallel complementarity. Group 3 is composed of networks which are topologically complementary, logically complementary but not self-dual nor series-parallel. Group 4 is composed of networks which are logically complementary but not topologically complementary nor self-dual nor series-parallel. A mux-based XOR2 network is an example of network from group 4. Group 5 is composed of networks which are topologically complementary, logically complementary and auto-dual, but series-parallel. Group 6 is composed of networks which are logically complementary and self-dual but not topologically complementary nor series-parallel complementary, even each of the planes is individually series-parallel. An alternative XOR3 implementation is a network from group 6. Group 7 is composed of cells that are auto-dual and are not complementary (series-parallel, topologically or logically). The Müller cell, widely used in asynchronous circuits, can

be considered as a group 7 network if the memory stage at the output is disregarded. Finally, group 8 is composed of cells that are neither auto-dual nor complementary (series-parallel, topologically or logically). A tri-state network is an example of group 8. Figure 3.10 shows an example of network that composes each group of the classification presented before.

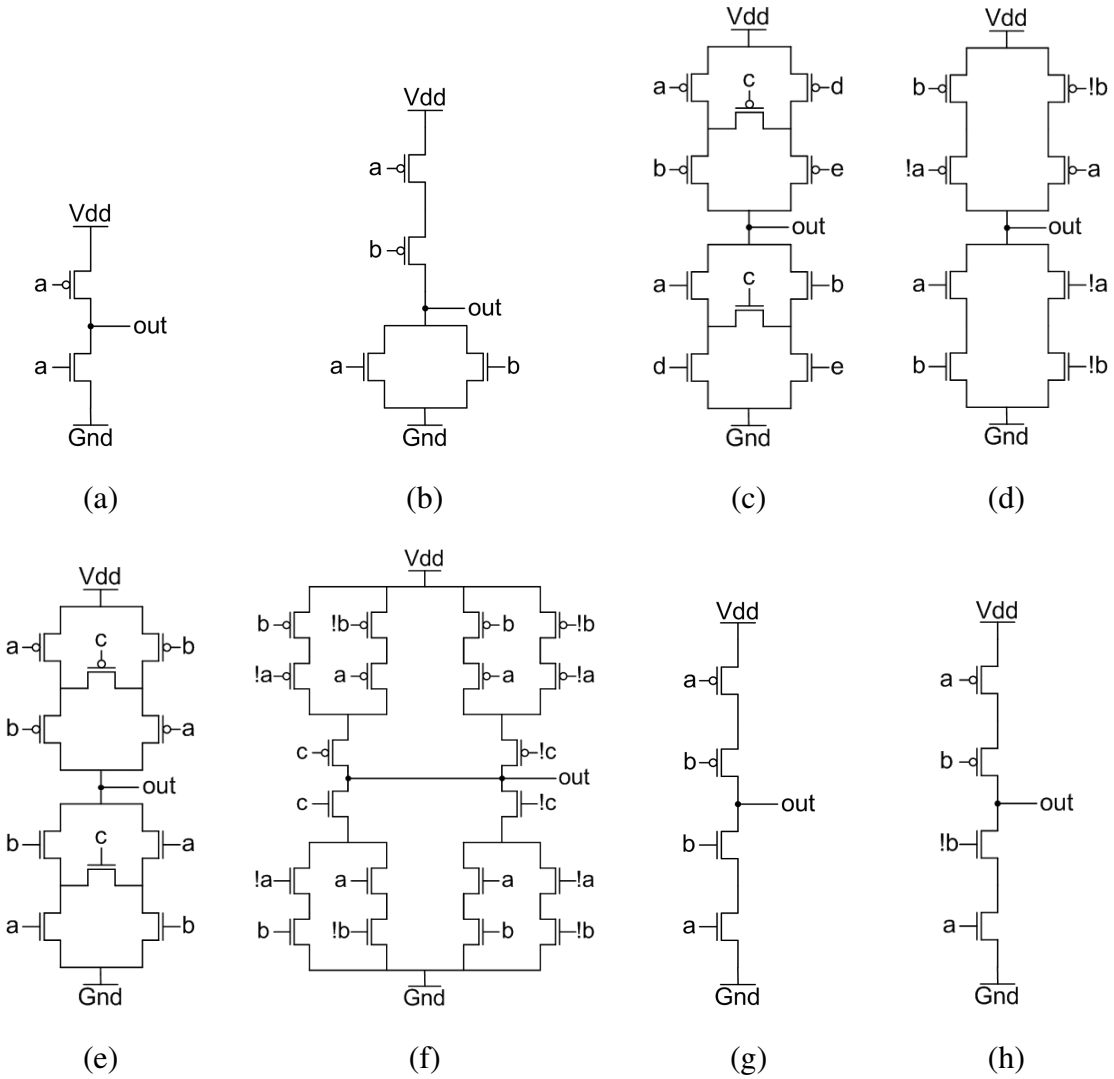


Figure 3.10: Networks from: (a) group 1, (b) group 2, (c) group 3, (d) group 4, (e) group 5, (f) group 6, (g) group 7, (h) group 8.

3.3 MOS Transistor as a Non-ideal Switch

As mentioned before, MOS (Metal Oxide Semiconductor) transistor is a logical switch capable of switch-on and to switch-off an electrical path according to the control signal applied to its ‘gate’ terminal. Transistors are built over a semiconductor substrate,

generally a Si-substrate. Two regions of the substrate contain a high concentration of ions, and are called ‘source’ and ‘drain’ terminals. These regions are separated by a channel under a strip of polysilicon, known as ‘gate’. Between the gate terminal and the substrate portion (‘bulk’) is inserted an insulator to avoid the direct contact of both. The gate terminal controls the ions induction in the bulk region, representing the portion between source and drain, allowing a current flowing through these regions. Figure 3.11 illustrates a MOS transistor structure.

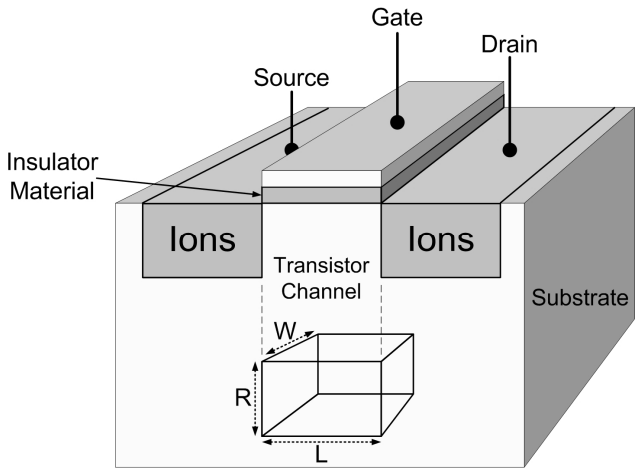


Figure 3.11: MOS transistor structure.

Two physical dimensions are of special interest in the MOS transistors. These dimensions are called **transistor width (W)** and **transistor length (L),** indicated in Figure 3.11. They are presented under the gate terminal, in the **channel** of the transistor. The main electrical characteristics of a MOS transistor are determined according to their dimensions (W and L) and the oxide thickness over the transistor channel.

Unfortunately, MOS transistor is not an ideal switch. These elements do not conduct when ideal logic *‘1’* or logic *‘0’* are applied to their gates. A behavior closer to the real functionality condition is that the gate voltage of a MOS transistor must present a given bias differential to the source terminal. This minimal voltage that allows the transistor conduction is known as **threshold voltage (Vth)**. The threshold voltage of PMOS and NMOS transistors may change according to the technology process and bulk potential. The important fact is that the transistors start or stop their conduction states when this voltage differential occurs in their terminals. Thus, the Vth affects the delay properties of the logic cell.

Another essential element that influences the logic cell performance is the **channel resistance (Ron)** when the transistor is conducting. The resistance definition of an electrical path may be expressed considering a current crossing a tri-dimensional conductor block, as shown in Figure 3.12. The larger is the block length, the larger is the resistance. On the other side, the larger is the block width, the smaller is the resistance. In a simple analysis, it could be modeled as:

$$Ron = (\rho * L) / (W * T) \qquad (3.6)$$

where ρ is the intrinsic resistance of the material that composes the block.

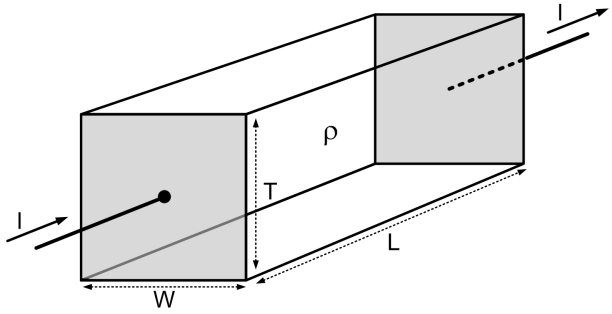


Figure 3.12: MOS transistor channel dimensions.

The MOS transistor channel can be analyzed as a tri-dimensional block (as illustrated in Figure 3.12). The designer can modify the transistor width and length to achieve the desired conduction behavior. However, it is necessary to respect the sizing constraints (minimum dimension for W and L) defined in the technology.

On the other hand, the gate terminal of the MOS transistor represents a capacitor. It can be viewed as a parallel plate capacitor with the gate on top and channel on bottom, with a thin oxide dielectric between. This situation collaborates for a non-ideal transistor behavior, since that **gate capacitance** influences the transistor operation. Most transistors used in logic design are of minimum manufacturable length because this results in highest speed and lowest power consumption. In addition to the gate, the source and the drain also represent capacitances. These capacitors are formed between the drain or source diffusion and the substrate, and are charged or discharged according to the bias condition over the transistor. These capacitances are not fundamental to the operation of the devices, but do impact circuit performance and hence are called **parasitic capacitors**. Notice that the drain and source capacitances are also dependent of the transistor width. The larger is the W, more capacitance will be present in the device.

All these elements (W, L, Ron, Vth, and capacitances) impact in the logic networks characteristics. Timing, power and area present different behaviors according to the variation of these elements in a given transistor arrangement. In a general way, by performing an adequate transistor sizing it is possible to achieve better networks implementation.

3.4 Transistor Sizing

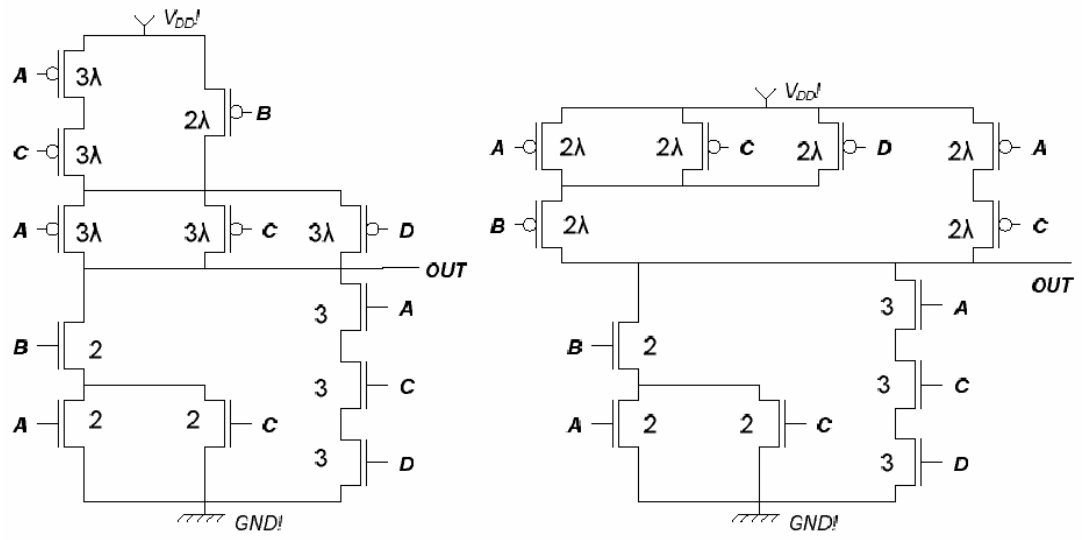
The designer of a VLSI circuit must consider not only functional correctness but timing behavior. Usually, there is some specification of how quickly the circuit must produce its output. Once a schematic, transistor-level description of the circuit is produced, it must be forced to meet the delay constraint. This is done by assigning sizes to the transistors.

Increasing the size of transistors in a VLSI circuit tends to decrease the delay through the circuit, but at the cost of increasing its area. While transistor area is usually only a small component of the total chip area, that is only because transistor sizes are usually reasonable. Minimizing delay can result in huge transistors. Beyond a certain point, however, larger transistors actually increase delay.

To perform the transistor sizing in this work, the Logical Effort method was implemented (SUTHERLAND, 1999). The next section discusses this method.

3.4.1 Logical Effort and Transistor Networks

The logical effort is a gain based method that allows to compare how costly it is for a given logic gate to compute the Boolean function it implements, comparatively to a reference inverter. This way, the gain across paths in a circuit are distributed evenly, and gates which do not have an high effort to compute logic (logical effort) will contribute with electrical effort, driving more significant output capacitances relative to their input. A straightforward method to compute the logical effort is described in (SUTHERLAND, 1999). It considers that the transistors in a complex gate have to be sized to have the same drive strength of a reference inverter. As an example, consider the circuits described in Figure 3.13a and Figure 3.13b, which implement the same logic function. The transistor sizes are shown on both figures, relative to a reference inverter where the NMOS transistor has size ‘1’ and the PMOS transistor has size ‘λ’. The logical effort for every input is the fanin (input capacitance) divided by the input capacitance of the reference inverter (1+λ). In the case of input ‘A’ the logical effort values are (5+6λ)/(1+λ) for Figure 3.13a and (5+4λ)/(1+λ) for Figure 3.13b. The circuit in Figure 3.13b has a reduced logic effort, as its pull up plane has less series transistors, which allow driving the same current with smaller transistors. Table 3.1 presents the logical effort values for both circuits. The sizing of the networks and the logical effort computations were done in accordance with the methods presented in (SUTHERLAND, 1999). The transistors sizing consider the transistor chains for every path between the output and the power source. The network in Figure 3.13b tends to have a smaller intrinsic delay when compared to the network of Figure 3.13a, as the transistors will have smaller sizes to deliver the same output current. This reduces parasitic capacitances and the intrinsic delay.



(a) (b)

Figure 3.13: Two circuits for the same logic function.

Table 3.1: Logical effort values for circuits in Figure 3.13.

Circuit	A	B	C	D	Total
Figure 3.13a	(5+6λ)/(1+λ)	(2+2λ)/(1+λ)	(5+6λ)/(1+λ)	(3+3λ)/(1+λ)	(15+17λ)/ (1+λ)
Figure 3.13b	(5+4λ)/(1+λ)	(2+2λ)/(1+λ)	(5+4λ)/(1+λ)	(3+2λ)/(1+λ)	(15+12λ)/ (1+λ)

3.5 Conclusions

This chapter discussed some alternative logic styles to the CMOS standard. A classification for two terminal disjoint networks was compiled in order to demonstrate the richness of the switch theory. Also, a discussion regarding the MOS transistor as a non-ideal switch element was done. Finally, the transistor sizing was explored and the logical effort method was presented.

4 ESTIMATION OF COSTS

To compare the different network implementations, some estimation methods can be applied. This chapter presents some methods used in this work to evaluate the networks. For delay evaluation, the Elmore delay model is used. For dynamic power dissipation, a method that considers the intrinsic capacitance of the transistor network is utilized. To evaluate the leakage behavior, this work makes use of three different leakage estimation models, which considers both gate and subthreshold leakage current. Finally, to evaluate area, a naïve and simple method is present.

4.1 Profile and Parameters Extraction

In order to evaluate a transistor network, it is needed to discover some information regarding the topology. This information is used by the estimation methods during the analyses or calculation process. Examples of this kind of information are:

- Number of nodes in a network;
- Number of transistors;
- Number of transistors connected per node;
- Number of branches;
- Number of paths between the output and the source nodes;
- Shortest path in a network;
- Larger path in a network;
- Size of transistor;
- Equivalent transistor size in a given path;
- Fanin;
- Distance between two nodes in a network;
- Etc.

Several algorithms were implemented to obtain all this information. Other necessary data to estimate the behavior of the networks are dependent of the technology process in which the transistor networks will be investigated. This information are used here as external parameters, obtained through Spice simulations. They are extracted once for a given technology process and saved in a parameter input file. So, this file is used by the estimation methods when necessary. Examples of this sort of data are:

- Channel resistance of PMOS and NMOS transistors;
- Drain and source capacitances (as function of the transistor width);
- Threshold voltage;
- *Vdd* voltage;
- Etc.

As it is not the focus of this work, the parameters extraction will not be discussed here. But it is important to know that they exist. More information about the parameters extraction may be found in the references of each estimation method.

4.2 Timing Estimation

The performance of CMOS circuits can be characterized by the time needed to charge and/or discharge the intrinsic capacitors of these circuits. In fact, the existence of parasite elements (capacitances and resistances) impacts directly in the electrical signal propagation on the circuits. Some definitions about time can be considered when analyzing a given logic circuit:

- Rise time (tr): It corresponds to the time needed to change the signal from ‘0’ logic to ‘1’ logic. This time is usually measured when the signal changes from 10% to 90% of its voltage variation in the output.
- Fall time (tf): It corresponds to the time needed to change the signal from ‘1’ logic to ‘0’ logic. This time is usually measured when the signal changes from 90% to 10% of its voltage variation in the output.
- Delay time (td): It corresponds to the maximum time from the input signal crossing 50% to the output signal crossing 50%. As this delay generally is not the same for ‘0’ to ‘1’ and for ‘1’ to ‘0’ transitions, it is common to separate it in td_{hl} (delay time high-to-low) and td_{lh} (delay time low-to-high).

According to Weste (2005), quick delay estimation is essential to designing critical paths of digital circuits. Although timing analyzers or circuit simulators can compute very detailed switching waveforms and accurately predict delay, good designers cannot be dependent on simulation alone. Also, simple models are important because they allow rapidly estimating delay, understanding its origin, and figuring out how it can be reduced.

One of the most used methods to estimate delay, the Elmore delay model (ELMORE, 1948), is based on the computation of the delay in an equivalent RC circuit. In this model, each transistor is modeled as a resistance between their source and drain terminals, and all parasite capacitances are modeled as grounded capacitances. Viewing 'on' transistors as resistors, it is possible to see that a chain of transistors can be represented as an RC ladder as shown in Figure 4.1.

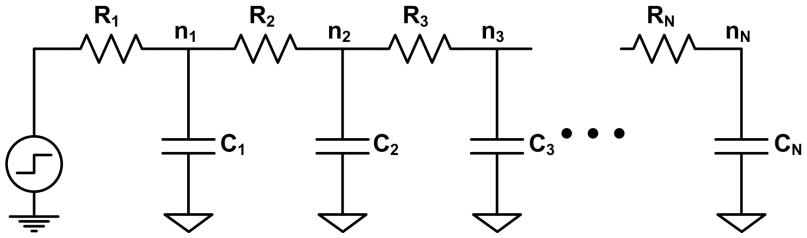


Figure 4.1: RC ladder for Elmore delay.

The Elmore delay model estimates the delay of an RC ladder as the sum over each node in the ladder of the resistance $R_{n\text{-}i}$ between that node and a supply multiplied by the capacitance on the node. The equation (4.1) models the Elmore delay.

$$\tau = \sum R_{n-i} * Ci \qquad (4.1)$$

It is known that the Elmore delay model represents a simple approximation of the actual delay, but, according to Rabaey (2005), it is acceptable for fast estimation. It offers the designer a powerful mechanism for providing a quick estimate of the delay of transistor networks.

The Elmore delay model, described in details and validated in (SCHNEIDER, 2004), was implemented in this work to compare the delay of different transistor networks implementation of same functions. The capacitances and resistances (for NMOS and PMOS transistors) are parameters used for the calculation. As mentioned before, these parameters are extracted through Spice simulation for a given technology process. The delay model considers the transistor width dependence during the calculation procedure. In this implementation, τ is calculated only for a single input signal variation. In other words, only one input variable is changed in each analysis.

4.3 Dynamic Power Estimation

Power dissipation is no longer a secondary issue in CMOS digital design (LIU, 1994). The increasing complexity and high-performance requirements of modern integrated circuits have led to high power consumption. Transistor level simulators with continuous-time modeling of the devices, like Spice, can be very expensive in terms of storage and computation time. Hence, a great effort has been devoted in the development of accurate analytical expressions power models (NAJM, 1994; BOGLIOLO, 1997; ALIOTO, 2007).

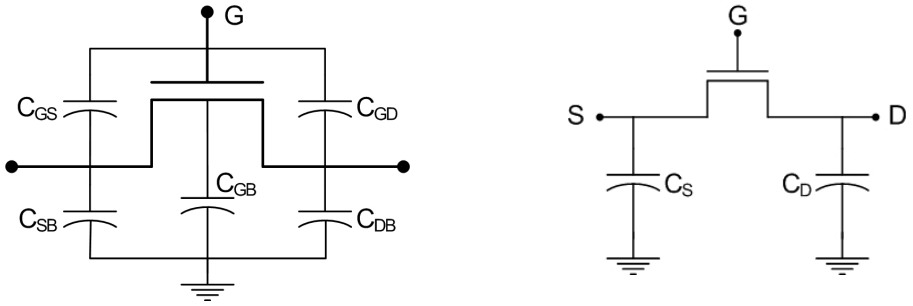
The dynamic switching power dissipation was the dominant factor compared to the other components of power dissipation in digital CMOS circuits for technologies down to 0.18 micrometers, where it is about 90% of total circuit dissipation (PARK, 2006). Short–circuit power is the second source of total power dissipation. During a transient on input signal, there will be a period in which both NMOS and PMOS transistor will conduct simultaneously, causing a current flow through the direct path existing between power supply and ground terminals. This effect usually happens for very small intervals. However, according to Veendrick (1984), this component represents less than 20% of the dynamic switching power consumption if the NMOS and PMOS transistors are sized in order to balance the rise/fall signal slopes at input and output nodes. Considering that, in this work only the dynamic switching power dissipation will be considered to investigate the dynamic power behavior of the networks.

Traditional gate-level power estimations are based on the simplified assumption that the supply current required by a CMOS circuit is essentially spent in charging load capacitances at outputs of the switching gates (TSUI, 1993). These output capacitances are mainly composed by the input capacitances of next interconnected gates. However, intrinsic capacitances also contribute for the power dissipation and cannot be neglected in the cell power estimation analysis, being a significant element in the cell power estimation analysis. In this context, a simplified analytical model to estimate intrinsic power consumption based on the charge required by intrinsic capacitances associated to a CMOS cell is presented by Chiappetta (2008).

The MOS capacitances can be divided in gate (*CG*), depletion (*CDB* and *CDS*) and overlap (*CGD* and *CGS*) capacitances, as shown in Figure 4.2a. However, a simplified model that considers only the intrinsic capacitance, illustrated in Figure 4.2b, is used. The drain and source capacitances are defined as follow:

$$C_D = C_{DB} + C_{GD} \tag{4.2}$$

$$C_S = C_{SB} + C_{GS} \tag{4.3}$$



(a) (b)

Figure 4.2: Capacitance model: (a) MOSFET and (b) simplified approach.

Disregarding the process variability, the drain and source area can be considered the same and, consequently, $C_{DB} = C_{SB} = C_{DEP}$.

In the proposed analysis, MOS transistor is evaluated in cutoff and saturation regions. According to Weste (2005), linear region is ignored since it is a transitory state and it does not compromise the model accuracy. According to Uyemura (1999), C_{GD} is considered always zero and C_{GS} is $2/3*C_G$ in saturation mode.

Based on previous statement, the intrinsic capacitance can be modeled as shown in Table 4.1.

Table 4.1: Intrinsic capacitances modeling.

Capacitance	Cutoff State	Saturation State
C_D	$C_{DEP}(w)$	$C_{DEP}(w)$
C_S	$C_{DEP}(w)$	$C_{DEP}(w) + 2/3*C_G(w)$

All capacitances are a linear function of the transistor width and are modeled as follow:

$$C(w) = A * w + B \qquad (4.4)$$

where, A and B are constant values extracted from electrical simulations using different transistor width.

The power dissipated by the intrinsic capacitances of a CMOS gate is the one used to charge them. The discharge current is supplied by the charge stored in the capacitances and should not be accounted in total power consumption. Considering the previous statement, the power dissipated by the intrinsic capacitances is the one when the output changes from '0' to '1'. The total intrinsic power consumption of a CMOS gate for a specific transition in the input vector is given by equation (4.5).

$$P = \sum (C_i) * V_{dd}^2 \qquad (4.5)$$

This method presented by Chiappetta (2008) was implemented to evaluate the intrinsic power consumption of logic gates. The steps of the algorithm consist in the follow:

- In a given input vector *x*, which leads the output to a logic '0' value, identify all transistor bias conditions.
- In a given input vector *y*, that changes the output state from '0 to '1', identify all transistor bias conditions.
- For each network node, calculate the node capacitance considering all transistors capacitances connected to this node.
- A sum of capacitance is done for all nodes that changed their logic states from '0' to '1' (charged nodes).
- The intrinsic power consumption is computed by multiplying this obtained capacitance (through the sum) by Vdd^2.

To validate the implemented method, a complex gate (Figure 4.3), sized with Logical Effort, was simulated in Hspice using the CMOS PTM 130 nanometers technology process at 80ºC. The simulated results are compared to the estimated ones in Figure 4.4.

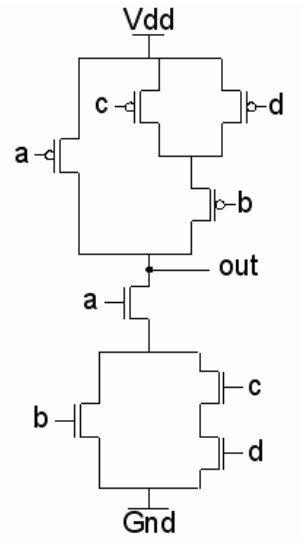


Figure 4.3: Complex gate.

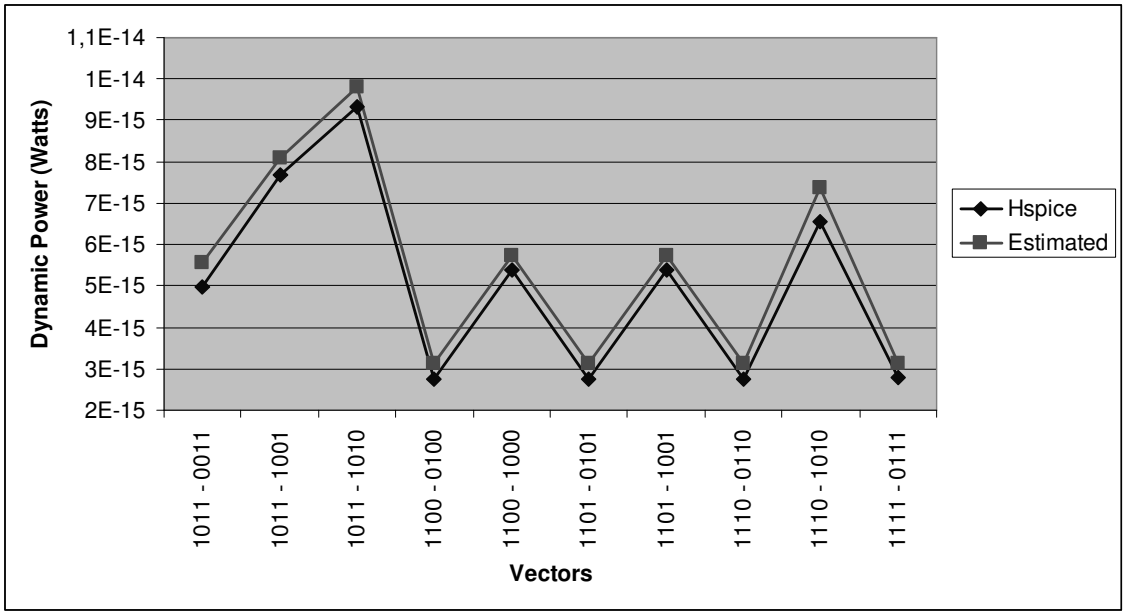


Figure 4.4: Hspice vs. estimated power consumption.

The obtained results present a difference from the Hspice simulations. However, for the purpose of this work, the model can be used to compare different cells implementing same logic functions, since it is capable of delivering an approximated behavior of the real values.

4.4 Static Power Estimation

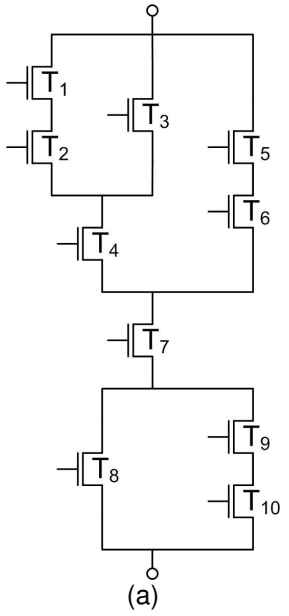
Static power consumption is nowadays a crucial design parameter in digital circuits due to emergent mobile products. Leakage currents, the main responsible for static power dissipation during idle mode, are increasing dramatically in sub-100 nanometers processes (ROADMAP, 2004). Subthreshold leakage rises due to threshold voltage scaling while gate leakage current increases due to scaling of oxide thickness (ROY, 2003).

To face this new challenge, a great effort has been done in developing models and estimators for design support. The ‘stack effect’ observed with off-transistor (i.e., devices that are turned off) in series arrangement is quite important for subthreshold current prediction (GU, 1996; CHENG, 1998). Differently from subthreshold leakage, gate oxide tunneling currents are observed in both on- and off-devices, according to the transistor biasing (RAO, 2003).

This section presents three different leakage estimation methods. The first one is dedicated to evaluate the subthreshold leakage only. The second is dedicated to evaluate the gate leakage. Finally, the third is an iterative and accurate method to estimate gate and subthreshold leakage in digital circuits.

4.4.1 A Simple Subthreshold Leakage Estimation

The idea of this method is based on the device electrical conductance association, that is, the conduction of parallel devices are summed while in series arrangements the equivalent conductance is inversely proportional to the number of devices. Being Gt[n] the conductance of the n-index transistor in the arrangement of Figure 4.5a, the equivalent conductance Geq is illustrated in Figure 4.5b.



(a)

$$Geq = \frac{K}{\frac{1}{\frac{1}{\frac{1}{Gt5}+\frac{1}{Gt6}}+\frac{1}{\frac{1}{Gt4}+\frac{1}{\frac{1}{Gt3}+\frac{1}{\frac{1}{Gt1}+\frac{1}{Gt2}}}}}+\frac{1}{Gt7}+\frac{1}{\frac{1}{Gt8}+\frac{1}{\frac{1}{Gt9}+\frac{1}{Gt10}}}}$$

(b)

Figure 4.5: (b) Equivalent conductance for the transistor network described in (a).

As discussed in (GU, 1996), in the case of series transistor the leakage reduction from a single off-device to two stacked off-transistors depends also on the fabrication process parameters. As a result, a constant K must be included in the last step of the calculation procedure in order to calibrate the final result. This K value is obtained by relating the leakage current of two-stack and single off-device configurations. In this sense, two constants Kn and Kp may be derived according to NMOS and PMOS arrangements, respectively.

Figure 4.7 illustrates the results obtained for this method over the transistor arrangement presented in Figure 4.6. The Hspice simulations were carried out by using the CMOS PTM 180 nanometers parameters at 80ºC.

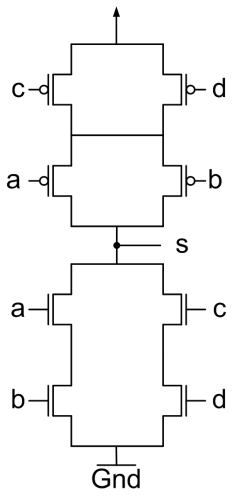


Figure 4.6: A 4-input transistor network.

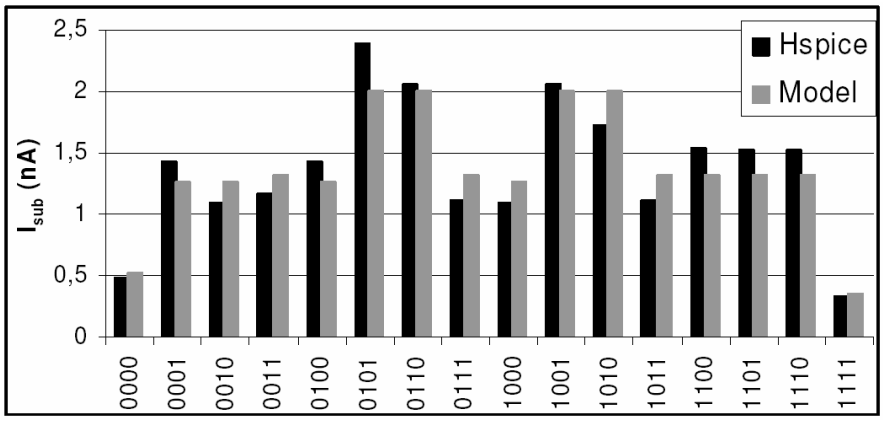


Figure 4.7: Subthreshold leakage currents in the CMOS structure from Figure 4.6, for each input vector.

As it can be seen, the correlation with Hspice presents non-accurate results. However, using this method it is possible to identify the input vector that produces less leakage consumption. If the main goal of the designer is only to find the appropriate vector to put the circuit in standby mode, so this approach may be useful.

Also, it is important to notice that this method is suitable for technology processes over 130 nanometers. When analyzing more recent technologies, as 90 or 65 nanometers, the gate leakage component is added to the total leakage consumption, making unfeasible the use of this method.

4.4.2 Gate Leakage Estimation

The gate leakage occurs when transistors are turned ON and OFF. Gate leakage current is independently in both, turned ON or OFF, transistor states. When transistor is turned OFF the current flows by the overlap source and drain regions. In the case where the transistor is turned ON, the current uses the overlap source/drain regions and the transistor channel. For these reasons, gate leakage is usually higher in such condition.

Considering previous statement, an easy method to investigate gate leakage current is evaluating the transistor bias conditions. Figure 4.8 presents all eight possible bias conditions for a NMOS transistor. Figure 4.8f and 4.8g can be ignored because they represent transient states and do not occur in steady state. In Figure 4.8a and 4.8h gate leakage is not present because all terminals have the same potential. In the other conditions gate leakage has to be computed.

Assuming that, the idea to compute gate leakage is very simple. For a given technology process the gate leakage for these transistor bias conditions are measured. So, when analyzing a transistor network, it is only necessary to discover the bias condition for each element in the network. The total gate leakage is the sum of the gate leakage of all transistors.

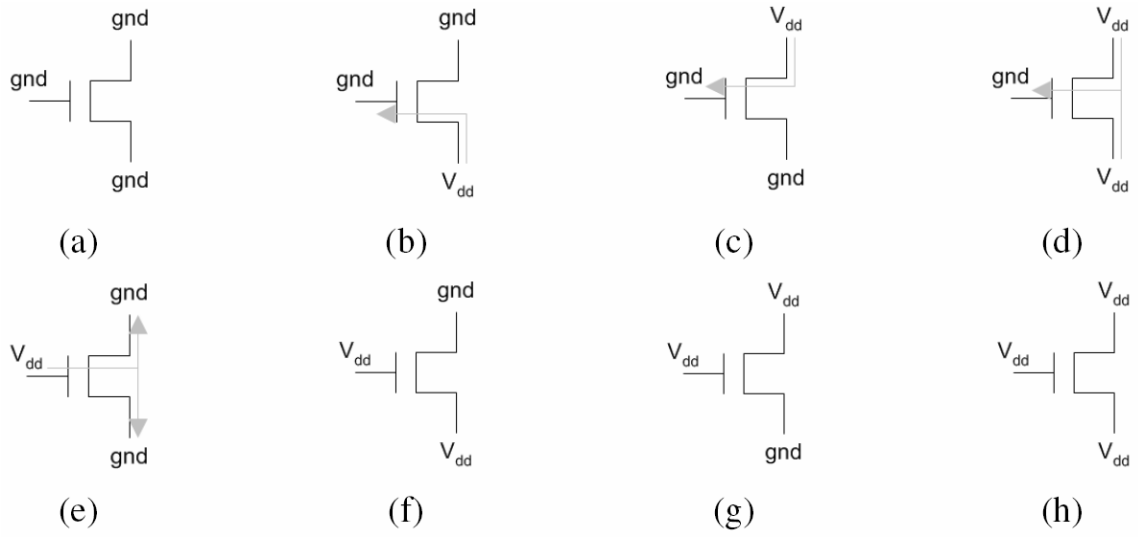


Figure 4.8: Possible bias condition for NMOS transistors in digital circuits.

4.4.3 Accurate Analytical Method for Static Current Estimation

The interaction among leakage mechanisms cannot be ignored in the analysis of static consumption. Some works in the literature evaluate separately the subthreshold and the gate components, leading to non-accurate results (CHENG, 1998; ROSSELLÓ, 2005; YANG, 2005). An iterative gate and subthreshold estimation method was

proposed by Butzen (2008) in order to delivery better results. The main advantage of this method is its capability of calculating leakage for logic cells with more than two logic levels. This approach was implemented here with some modifications to make possible its application in Wheatstone-bridge networks.

In the terminology used in this solution, *on-plane* and *off-plane* indicate the conducting and not conducting planes, respectively. From the *off-plane* it is extracted the *off-network* which represents the actual electrical circuit responsible for isolating the supply (*Vdd/Gnd*) to the output terminal.

Subthreshold and gate leakage currents are modeled by equation (4.6) and (4.7), respectively, where *W* is the transistor width and *VT* is the thermal voltage. *Vgs, Vds* and *Vbs* are respectively the gate-, drain- and bulk-to-source voltages. The terms *IS0, η, γ, n, Ig0* and *K* are constant values extracted from electrical simulation in a pre-characterization procedure of the target technology.

$$I_S = I_{S0} W e^{\frac{V_{gs}+\eta V_{ds}+\gamma V_{bs}}{nV_T}} \left[1 - e^{\frac{-V_{ds}}{V_T}}\right] \tag{4.6}$$

$$I_g = I_{g0}.W.e^{\frac{-K}{|V_{gs}|}} \tag{4.7}$$

The steps of the implemented algorithm are:

a) Identification of the off-plane, according to the input logic combination or the output logic level.

b) Extraction of the off-network from the off-plane considering the on/off devices status. On-devices are short-circuited and replaced by current sources representing the gate leakage current contribution of the transistor. Notice that, when on-devices short-circuit internal nodes eventually parallel off-devices and transistor clusters (sub-networks) are removed. In this case, for each removed device a respective current source must be added at this node to maintain the effect of its gate leakage in the total leakage calculation. Moreover, a voltage drop ΔV throw on-devices connected to the output node is observed and they cannot be considered as ideal short-circuits. It means that the other device terminal assume $Vdd-\Delta V$ in the case of NMOS pull-down off-plane and $Gnd+\Delta V$ for the PMOS pull-up off-plane.

c) Identification of the DC polarity (biasing) of each off-device present in the off-network. It is a straightforward task when treating purely series-parallel transistor arrangements. In the case of non-series-parallel

configurations, a procedure has to be performed taking into account the distance of transistor drain and source terminals to supply and output terminals. The distance here is understood as the number of off-devices in the shortest path to reach the network terminals (*Vdd/Gnd* supply and output). When the arrangement is symmetric, it is decided randomly with negligible loss in accuracy.

d) Ordering of the internal nodes in the off-network with unknown voltages. This ordering is done according to the internal nodes distance to the output node. Again, for nodes with the same distance to the output terminal, the distance to the supply terminal is considered, giving priority to the node far from that. In the case of symmetric arrangements this choice is random.

e) Calculation of the drain-to-source voltage (*Vds*) of each transistor by applying the Kirchhoff's Current Law (*KCL*) at each internal node. All subthreshold and gate leakage currents related to each node are taken into account. Differently from numerical solvers, like electrical simulators, the purpose here is to calculate in the predefined order of step (d). All unknown node voltages are temporarily considered as ground or power voltages, for pull-down NMOS and pull-up PMOS off-planes, respectively.

f) Definition of the voltage at each node based on the transistor *Vds* voltages, previously calculated in step (e). Starting from supply *Vdd/Gnd* terminal, compute the voltage of each unknown node summing the *Vds* value of each transistor in the path node-supply, respecting the inverse node order established in step (d). In the case that the node has more than one possible voltage value, i.e. there is more than one path to reach the supply terminal; the node potential is determined by the highest value obtained.

g) Estimation of the total leakage current considering the internal node voltages previously determined in step (f). It corresponds to the sum of all leakage currents flowing from the *Vdd* terminal or to the *Gnd* one. For instance, consider the second option, i.e. the currents flowing to *Gnd*. The total leakage is given by the sum of the subthreshold current of all transistors connected directly to *Gnd* terminal, the gate leakage of on- and off-devices in the off-plane, and the gate leakage of on-devices in the on-plane.

To validate the implemented method, 42 logic gates extracted from Genlib 44-6, with up to six inputs, were evaluated. Results obtained with the proposed method were compared against Hspice and against the method presented in (YANG, 2005), where subthreshold and gate oxide currents are evaluated separately and then summed.

This comparison is depicted in Figure 4.9. The electrical simulations were carried out by using the CMOS PTM 45nm parameters (ZHAO, 2006), at 80ºC.

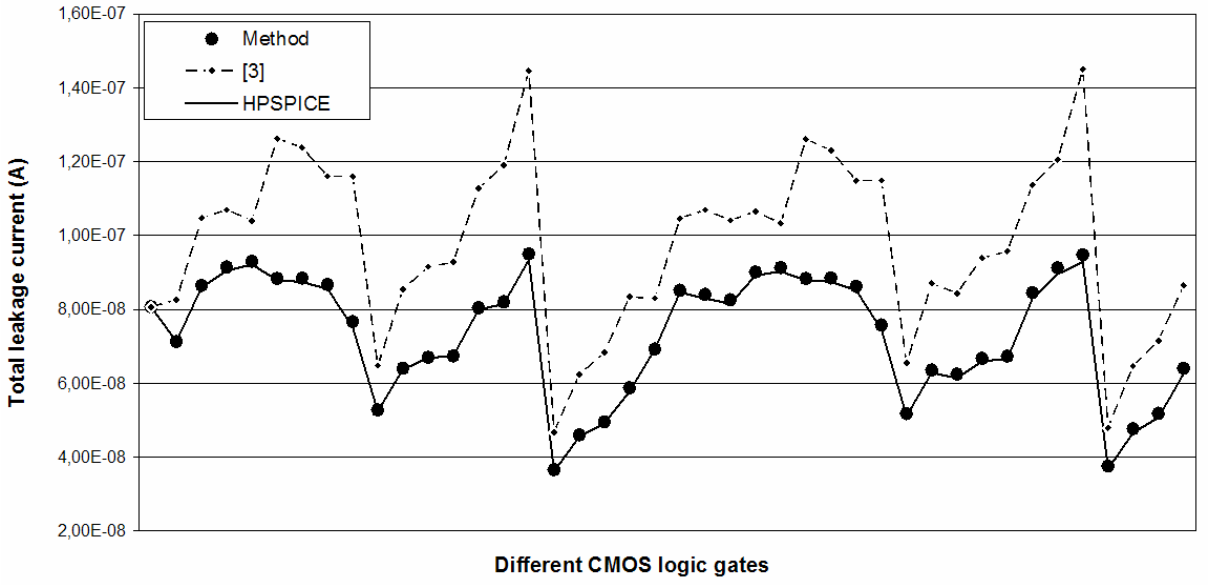


Figure 4.9: Total leakage estimation comparison for different CMOS gates.

4.5 Area Estimation

When regarding area, it is important to utilize logic cells with small layout design to guarantee better implementations of digital circuits. In order to achieve such a goal, it is desirable that the transistors composing the logic planes of a given logic cell can be aligned. Such a situation would eliminate the need for unnecessary internal connections between the transistor gates, possibly minimizing cell dimensions.

In this context, a solution is presented to achieve networks with maximal matching between transistor gates at a symbolic layout (topological level). Furthermore, a naïve calculation procedure estimates the layout width using design rules extracted from the technology process. This approach cannot delivery the exact layout area as the internal routing is not evaluated. However, it is capable of delivering good information when comparing different layout implementation.

4.5.1 Searching Eulerian Paths

In graph theory, Eulerian paths are paths that visit each edge in a graph exactly once. They were first discussed by Leonhard Euler while solving the famous problem of the Seven Bridges of Königsberg in 1736 (EVEN, 1979). Graphs containing such paths

are called traversable. A graph is traversable when it contains zero or two vertices of odd degree (DROZDEL, 2002). Fleury's algorithm (UEHARA, 1981) is widely used for searching Eulerian paths in traversable graphs. In short, the algorithm involves starting from one of the two odd vertices and traversing the graph, crossing all edges only once and finally arriving at the other odd vertex. If there are no odd vertices, any vertex can be used as a starting point. In modern microelectronics, this concept is very important, since a network of transistors can be represented as a multigraph where Eulerian paths may be used to define the positioning of transistors in a layout implementation.

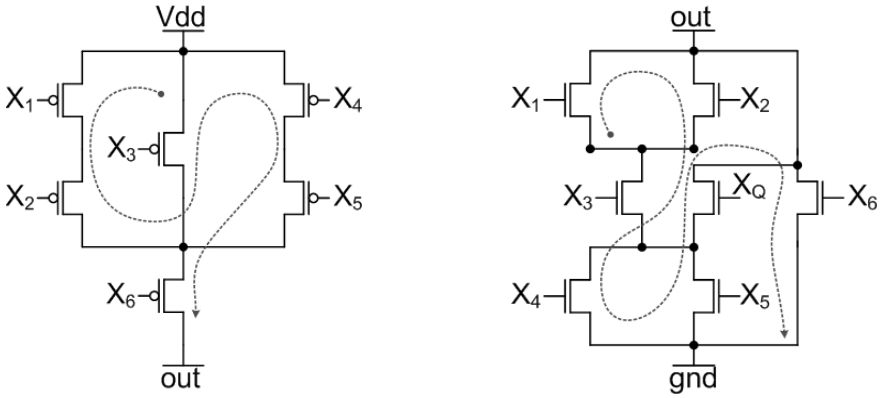


Figure 4.10: (a) PMOS transistor network and (b) NMOS transistor network showing possible Eulerian paths.

When a graph contains more than two vertices of odd degree, dummy edges may be inserted between them, making their degrees even. If enough dummy edges are inserted, any connected graph can be made traversable. Figure 4.10b illustrates the insertion of a dummy transistor (XQ) in a NMOS transistor network containing four nodes of odd degree. Note that the dummy transistor can be inserted between any pair of odd nodes.

Given a transistor network containing 'n' nodes of odd degree ($n > 0$), the number of dummy transistors required to make it traversable (d) can be obtained from the equation $d = (n - 2)/2$. For the example illustrated in Figure 4.10b, only one dummy edge is necessary.

A given logic plane in a disjoint transistor network may contain several Eulerian paths. In order to find all possible paths, the following steps are applied:

1. Each logic plane in the transistor network is converted into a multigraph representation.
2. The number of dummy edges to be inserted in the graph is determined using the equation described above.

3. Dummy edges are inserted between all possible pairs of odd vertices.
4. The multigraph is then traversed, starting at each of the odd vertices (or all the vertices, if there are none). The number of dummy edges used in a path is limited to the amount obtained in step 2.
5. All Eulerian paths found are stored in a tree-like structure to be analyzed by the gate matching algorithm. The tree nodes represent the gates of transistors in the network, and paths between the root and the leaves represent Eulerian paths. Figure 4.11 shows partial path trees for each of the logic planes illustrated in Figure 4.10.

4.5.2 Gate Matching

The gate matching process consists on finding a pair of Eulerian paths – one for the NMOS plane and one for the PMOS plane of the same transistor network – containing the same sequence of transistor controlling signals. It is important because aligning gates reduces the complexity of internal connections between the NMOS and PMOS planes. In this context, a good match could benefit the routing procedure, which is one of the most critical steps when generating a cell layout implementation. In addition, the layout area requirements could be minimized, since there is no need for extra rows to connect the crossing polysilicon gates. This leads to a smaller layout implementation, and avoids the use of an extra layer of metal in order to connect unaligned gates. Figure 4.12 illustrates two possible symbolic layout solutions (aligned and unaligned) for the cell shown in Figure 4.10.

To achieve gate matching, the following algorithm is proposed:

- Two trees obtained as described in Section 4.5.1 are simultaneously traversed in a recursive manner, starting at their roots.
- Each node in a tree is compared to its counterpart in the other tree. If a given node does not exist in one of the trees, it is removed, along with its child nodes.
- At the end of the algorithm, only corresponding nodes remain. These nodes represent matching gates in a pair of Eulerian paths.

Figure 4.11c illustrates the partial tree for the cell described in Figure 4.10 after the gate matching algorithm has concluded. Note that only one tree is shown, since the two resulting trees are identical.

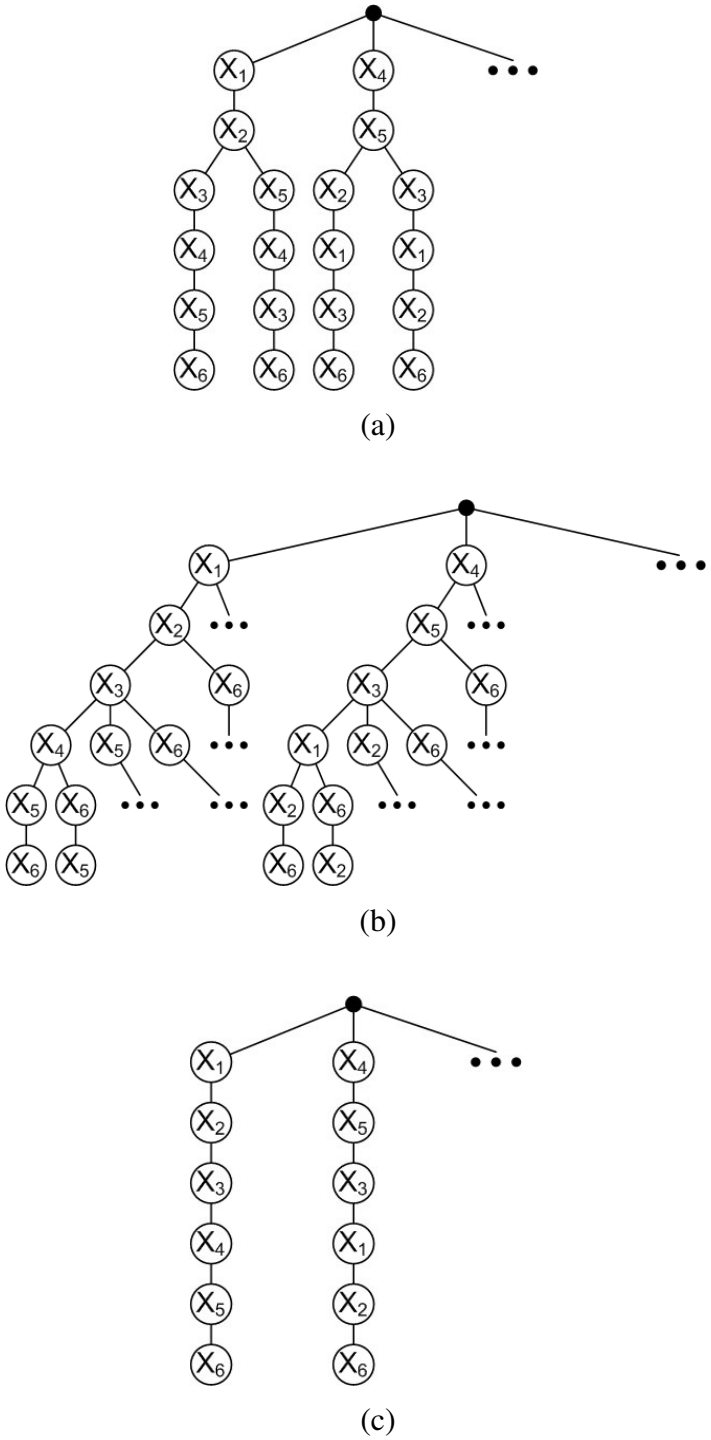


Figure 4.11: Partial tree for the cell in Figure 4.10, before (a, b) and after (c) the gate matching algorithm.

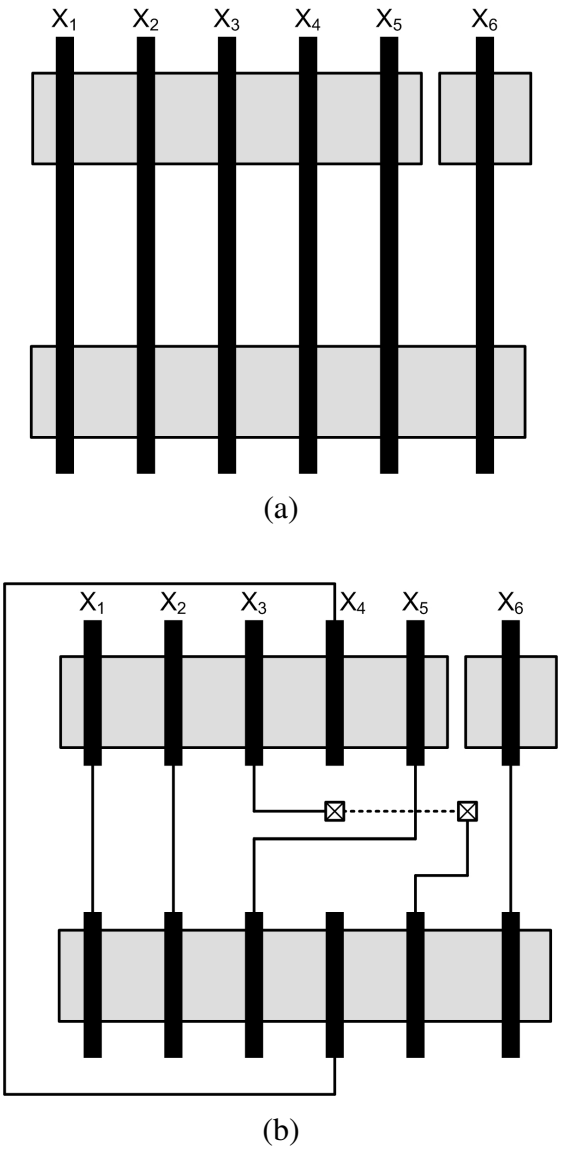


Figure 4.12: Two possible symbolic layouts for the cell in Figure 4.10, showing matched (a) and mismatched gates (b).

4.5.3 Width and Area Estimation

Once a gate matching is defined it is possible to evaluate the area width. The first step consists in to extract five relevant distances from the technology process documentation. These distances are the following:

1. Distance from polysilicon to the left diffusion area, considering contact;
2. Distance from polysilicon to the right diffusion area, considering contact;
3. Distance from channel to channel, considering contact;
4. Distance from channel to channel, disregarding contact;

5. Distance from channel to channel, considering a break.

Figure 4.13 illustrates the distances described above. All these distances will feed the calculation procedure, as it will be seen in the sequence.

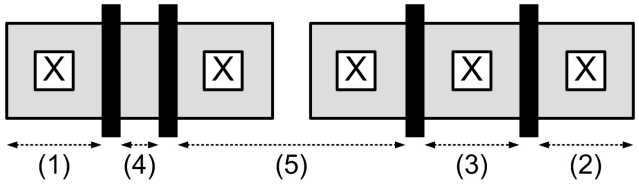


Figure 4.13: Relevant distances extracted from technology process documentation.

In a second step, by analyzing the Spice netlist cell description, the number of occurrence of contacts, breaks and transistor gates are annotated for each logic plane (PMOS and NMOS).

Finally, using these information and the distances previously obtained from the technology process documentation, it is possible to calculate the cell width by multiplying the distance values and the occurrence information.

Notice that the cell height is not investigated. So, in order to calculate the cell area, it is also necessary to set a cell height value. In the practice, thinking about logic libraries, all cells present the same height. For the purposes of comparing two different area implementations it is possible to set a same arbitrary height value for both. The estimated values may present a huge difference from the real area values. However, the comparison between the cell implementations is still valid and may be used as criteria to choose the smallest one.

4.5.4 Validating Area Estimation

To validate the area estimation technique a subset of cells extracted from Genlib 44-6 up to 4 inputs was used. All logic functions were implemented in CSP logic style. The UMC 130 nanometers technology process was utilized. To generate layout for these cells, the Nangate Cell Generator (NANGATE, 2008) was used. This commercial tool accepts transistor netlist description as input and delivers the final layout of the cells.

The used distances to estimate the cell width are described in Table 4.2. The cells height was fixed in 5.33 micrometers to compute the area. Figure 4.14 presents the results obtained for the real layout and the estimated one.

Table 4.2: Distances used to validate the area estimation.

Distance	Value
1	0.51 µm
2	0.51 µm
3	0.51 µm
4	0.36 µm
5	1.02 µm

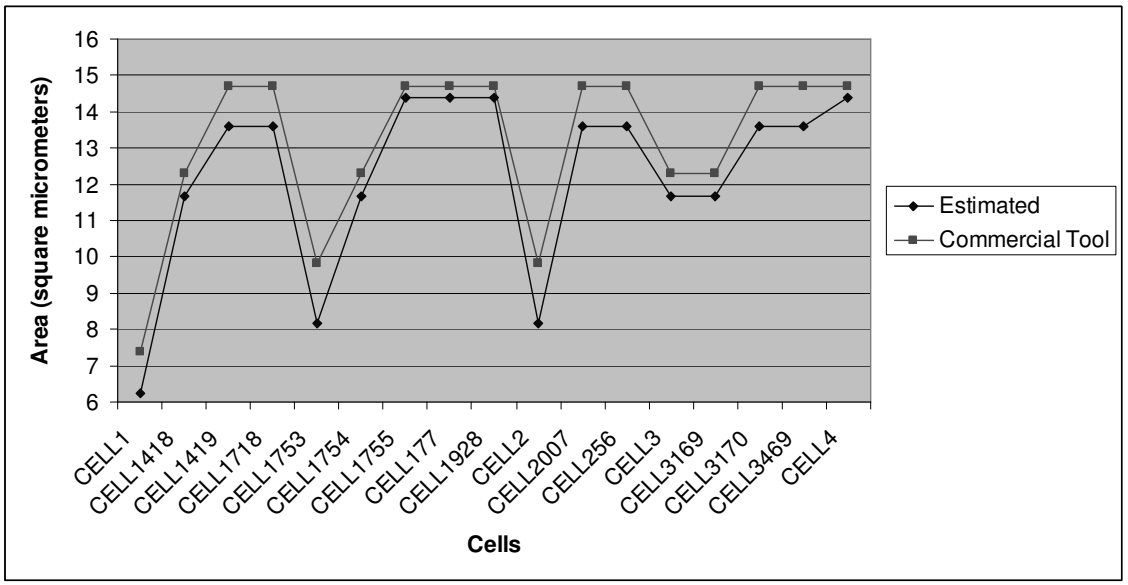


Figure 4.14: Results for the validation of the area estimation.

As mentioned before, the values are quite different. However, the area values obtained through the estimation present a good correlation with the real generated layout. This strong correlation permits to use this area estimation technique to compare different logic cells implementation, since it gives a good idea about the silicon needs of these cells.

4.6 Conclusions

This chapter presented some estimation methods to investigate area, dynamic consumption, delay, and leakage behavior of different logic cells implementation. All methods demonstrated a good an acceptable result approximation of the real values. The iterative gate and subthreshold leakage estimation method is an exception. This method presents a very accurate correlation with Hspice results as demonstrated in the validation experiment.

5 EXPERIMENTAL RESULTS

This chapter presents the experimental results for different sets of logic functions. All logic functions were implemented in the following logic styles:

- CSP
- NCSP
- BDD
- OpBDD
- LBBDD

After automatically generated, all transistor networks were sized using the logical effort method. The minimal transistor width for NMOS and PMOS transistors was 0.3 and 0.6 micrometers respectively.

The idea of using diverse sets of logic functions is to make possible to analyze the electrical and physical behavior of these cells when implemented in different logic styles. To compare the cells implementation, the estimation methods described in chapter 4 were used. For investigating the leakage behavior, the iterative gate and subthreshold leakage estimation method, presented in Section 4.4.3, was applied. For delay, the Elmore delay model was used. For area analysis all cells were set with a height of 5.33 micrometers (dimension used for cells with 13 rows in library containers). Finally, to evaluate the power consumption, the intrinsic power consumption method, presented in Section 4.3, was utilized.

Area, delay and dynamic consumption were evaluated over transistor networks using 130 nanometer technology. For leakage evaluation, 45 nanometers technology was used. It is due to the fact that 130 nanometer processes do not consider gate leakage current. Therefore, the parameters used for estimating area, delay and dynamic consumption were extracted using a CMOS 130 nanometers technology process. For leakage analysis, the parameters were extracted from PTM 45 nanometers technology process.

The following sections will discuss the results obtained for each set of logic functions.

5.1 Results for Genlib 44-6 up to 4-input

The first set of logic functions was extracted from the Genlib 44-6 cell library. All logic functions up to 4-input were implemented in the different logic styles previously described. These logic functions are negative unate, that means all input variables in the transistor networks are in just one polarity. All equations are originally factorized, and they do not present repeated literals in their description. The 44-6 information means that all functions, when implemented in CSP logic, present up to 4 transistors in series, 4 parallel branches, and no more than 6 logic levels. A particularity of this set is the lack of XOR functions.

As mentioned before, the number of transistor in series directly affects the delay characteristics of logic cells. In this sense, it is expected that NCSP and LBBDD logics present better results in terms of delay. However, when generating transistor networks from unate functions, the OpBDD logic style may deliver the same networks than LBBDD, since the optimization process is basically similar. In fact, LBBDD networks are optimizations over the OpBDD ones. As the OpBDD achieves the minimal network implementations, the LBBDD generation process cannot get more optimized transistor arrangements.

Another important point concerns CSP and NCSP. For unate functions without repeated literals, NCSP logic cannot obtain benefits in terms of transistor count and number of transistor in series. A CSP network is the best implementation for an unate logic function when it does not present repeated literals and it is the minimum literal cost factorized form. Thus, NCSP logic will deliver the same network.

This way, for the Genlib 44-6 logic functions, CSP, NCSP, OpBDD and LBBDD will generate the same transistor networks. The only difference in these networks could be the transistor order in the internal arrangements. However, using a BDD ordering or a structural ordering procedure, it is possible to reach exactly equal networks.

Table 5.1 shows the obtained average delay results for each logic cell. The values are in seconds. As it can be seen, CSP, NCSP, OpBDD and LBBDD present same delay values. BDD networks present worst results because they have more transistors than the other networks.

Notice that the Cell0 presents the same delay for all logic styles. This cell is an inverter.

Table 5.1: Average delay results (in seconds) for Genlib 44-6 up to 4-input.

	CSP / NCSP / OpBDD / LBBDD	BDD
Cell0	3,53E-12	3,53E-12
Cell1	1,79E-11	4,53E-11
Cell2	2,82E-11	4,56E-11
Cell3	3,76E-11	1,77E-10
Cell4	6,08E-11	1,93E-10
Cell5	4,10E-11	2,20E-10
Cell6	1,15E-10	2,04E-10
Cell7	6,65E-11	7,04E-10
Cell8	7,08E-11	5,10E-10
Cell9	7,57E-11	4,33E-10
Cell10	9,55E-11	5,47E-10
Cell11	1,64E-10	5,71E-10
Cell12	1,31E-10	6,41E-10
Cell13	5,89E-11	5,69E-10
Cell14	7,54E-11	5,23E-10
Cell15	1,80E-10	7,15E-10
Cell16	3,34E-10	6,24E-10

Table 5.2 shows the average intrinsic power consumption obtained for this set of cells. The values are in Watts. As expected, the results are similar to those obtained in the delay analysis. CSP, NCSP, OpBDD and LBBDD present the same power consumption behavior.

In Table 5.3, the estimated leakage currents are shown. The results are in Amperes and represent the average value. Notice that these results are for 45 nanometers technology process, differently than the dynamic power consumption that was obtained considering the 130 nanometers technology.

Once more, the leakage values obtained for CSP, NCSP, OpBDD and LBBDD are equivalent. Transistor stacks in BDD networks tend to be larger. From a static consumption point of view it is good, because the greater is the transistor stack, the smaller is the leakage current flowing in the network.

Table 5.2: Average power consumption (in Watts) for Genlib 44-6 up to 4-input.

	CSP / NCSP / OpBDD / LBBDD	BDD
Cell0	3,88E-16	3,88E-16
Cell1	1,41E-15	2,18E-15
Cell2	1,29E-15	2,04E-15
Cell3	2,06E-15	6,44E-15
Cell4	3,03E-15	5,07E-15
Cell5	2,83E-15	6,44E-15
Cell6	2,69E-15	4,85E-15
Cell7	4,99E-15	1,40E-14
Cell8	4,48E-15	1,31E-14
Cell9	3,42E-15	9,53E-15
Cell10	4,84E-15	1,18E-14
Cell11	5,33E-15	9,23E-15
Cell12	4,86E-15	1,37E-14
Cell13	4,77E-15	1,31E-14
Cell14	3,54E-15	9,91E-15
Cell15	4,79E-15	1,18E-14
Cell16	4,62E-15	8,92E-15

Table 5.3: Average leakage current (in Amperes) for Genlib 44-6 up to 4-input.

	CSP / NCSP / OpBDD / LBBDD	BDD
Cell0	8,07E-08	8,07E-08
Cell1	1,02E-07	7,54E-08
Cell2	9,62E-08	8,51E-08
Cell3	1,29E-07	1,41E-07
Cell4	9,69E-08	9,28E-08
Cell5	1,26E-07	1,48E-07
Cell6	8,57E-08	1,00E-07
Cell7	1,50E-07	1,96E-07
Cell8	1,29E-07	2,13E-07
Cell9	1,77E-07	2,78E-07
Cell10	1,24E-07	1,59E-07
Cell11	8,56E-08	8,52E-08
Cell12	1,43E-07	2,18E-07
Cell13	1,31E-07	2,30E-07
Cell14	1,77E-07	2,78E-07
Cell15	1,14E-07	1,77E-07
Cell16	6,92E-08	1,05E-07

Finally, Table 5.4 presents the results about area for the set of logic functions extracted from Genlib 44-6. The values are in square micrometers.

As previously expected, CSP, NCSP, OpBDD, and LBBDD presented equal area results. BDD networks present a high area penalty. In this logic, all transistor

associated to the BDD edges are available in the network. This fact collaborates to increase the network areas if comparing to OpBDD and LBBDD logics.

Table 5.4: Area results (in square micrometers) obtained for Genlib 44-6 up to 4-input.

	CSP / NCSP / OpBDD / LBBDD	BDD
Cell0	6,23	6,23
Cell1	8,15	13,59
Cell2	8,15	13,59
Cell3	11,67	26,86
Cell4	11,67	26,06
Cell5	11,67	26,86
Cell6	11,67	26,06
Cell7	14,39	38,53
Cell8	14,39	38,53
Cell9	13,59	35,01
Cell10	13,59	34,21
Cell11	13,59	37,73
Cell12	14,39	38,53
Cell13	14,39	38,53
Cell14	13,59	35,01
Cell15	13,59	34,21
Cell16	13,59	37,73

5.2 Results for Additional Logic Cells of a Library Container

As mentioned in the previous section, the Genlib 44-6 library does not contains XOR-like cells. These kinds of cells are binate, since they present the variables in negative and positive polarities. Commercial libraries generally have these cells implemented in different logic style than CSP. It occurs because XOR4 cells, for instance, when implemented in CSP delivery more than 4 transistors in series. So, XOR4 in CSP logic style is unfeasible. NCSP and LBBDD make possible the implementation of XOR4.

Another interesting cell to be implemented is the Cout function of a full adder. This function is unate, and it presents repeated literals in its description. Also, Genlib 44-6 library does not contain this logic function.

The next sub sections investigate these logic functions implementation. For example, a designer may add the best achieved networks of these functions to expand the library cell.

5.2.1 XOR Logic Functions

To evaluate XOR-like logic functions, three XOR were considered: XOR2, XOR3 and XOR4. These functions were factorized in order to deliver the most optimized transistor network. Table 5.5 shows the total transistor count for these cells in each logic style, disregarding the inverters needed to feed the complementary inputs.

Table 5.5: Transistor count for XOR logic functions.

	CSP / NCSP	**BDD / OpBDD / LBBDD**
XOR2	8	8
XOR3	20	16
XOR4	44	24

Table 5.6 shows the number of transistor ion series in both planes of each implementation.

Table 5.6: Transistor in series for XOR logic functions.

	CSP		**NCSP / BDD / OpBDD / LBBDD**	
	PU	**PD**	**PU**	**PD**
XOR2	2	2	2	2
XOR3	3	4	3	3
XOR4	4	8	4	4

The CSP and NCSP networks implementation present exactly the same transistor count. However, the CSP has a large number of transistors in series in one of logic planes. This occurs because one plane is the dual of the other. In the NCSP logic, PU and PD planes are generated from the on- and off-set equations respectively. Thus, both planes present the same size in the transistor stacks.

BDD, OpBDD and LBBDD networks are equal. The generation algorithm for these networks cannot achieve small networks. All transistors in the network derived from BDD are necessary and cannot be removed. This occurs because the XOR logic function is binate in all variables. Appendix B presents the schematic representations for these switch networks.

Table 5.7 shows the average delay obtained for the networks. All values are shown in seconds. XOR2, as expected, presents equal values for all implementations. XOR3 is better when using NCSP or BDDs implementations. For XOR4 logic function, the best choice is the BDDs implementation, which present smaller transistor stack and transistor count. As predictable, CSP is the worst implementation due to the larger transistor stack presented in one of the logic planes.

Table 5.7: Average delay (in seconds) for XOR logic functions.

	CSP	NCSP	BDD / OpBDD / LBBDD
XOR2	7,34E-11	7,34E-11	7,34E-11
XOR3	5,38E-10	3,45E-10	3,43E-10
XOR4	3,14E-09	1,14E-09	1,07E-09

Table 5.8 shows the dynamic power consumption. Like the results obtained for delay, it is possible to see similar behavior in this analysis. CSP has same transistor count than NCSP. However, the number of transistors per node in CSP is larger than NCSP. It means more capacitance per node, leading to more dynamic consumption.

Table 5.8: Dynamic consumption (in Watts) for XOR logic functions.

	CSP	NCSP	BDD / OpBDD / LBBDD
XOR2	2,69E-15	2,69E-15	2,69E-15
XOR3	1,23E-14	7,00E-15	6,68E-15
XOR4	4,55E-14	1,60E-14	1,26E-14

In Table 5.9, the achieved leakage current results are shown. The results are in Amperes. In terms of minimum leakage, CSP presents the smallest value. It is due to the fact that CSP has the largest transistor stack (greater is the stack, smaller is the leakage).

Table 5.9: Leakage current (in Amperes) for XOR logic functions.

	CSP	NCSP	BDD / OpBDD / LBBDD
XOR2	2,26E-07	2,26E-07	2,26E-07
XOR3	5,04E-07	4,52E-07	5,22E-07
XOR4	1,15E-06	8,06E-07	9,53E-07

Table 5.10: Area results (in square micrometers) for XOR logic functions.

	CSP	NCSP	BDD / OpBDD / LBBDD
XOR2	18,22	18,22	18,22
XOR3	43,49	43,49	42,21
XOR4	82,66	82,66	58,36

Table 5.10 illustrates the results in terms of area. The values are depicted in square micrometers. The estimated area results demonstrate that the transistor count may be used do give a good idea about the cell area. In this case, the fixed height is sufficiently enough to guarantee that internal network may be routed without performing transistor folding or enlarging the cell width.

5.2.2 Cout Function of a Full Adder

This logic function is very interesting to be analyzed. This cell is widely used in arithmetical circuit implementations. The Cout function can be expressed as *cout* = *a*b + a*c + b*c*. As this cell appears several times in regular adders and multipliers, it is important to use the most optimized version as it possible. Appendix C shows the transistor schematics for this logic function implemented in the target logic styles.

Table 5.11 presents the delay results. Table 5.12 presents the power consumption results. In Table 5.13 the leakage current is shown. At last, the obtained areas are described in Table 5.14.

Table 5.11: Delay results (in seconds) for Cout function of a full adder.

	CSP	**NCSP**	**BDD**	**OpBDD**	**LBBDD**
CoutFA	1,00E-10	8,26E-11	2,83E-10	1,42E-10	8,26E-11

Table 5.12: Dynamic consumption (in Watts) for Cout function of a full adder.

	CSP	**NCSP**	**BDD**	**OpBDD**	**LBBDD**
CoutFA	4,40E-15	3,77E-15	6,25E-15	5,80E-15	3,77E-15

Table 5.13: Leakage current (in Amperes) for Cout function of a full adder.

	CSP	**NCSP**	**BDD**	**OpBDD**	**LBBDD**
CoutFA	2,10E-07	1,83E-07	2,42E-07	2,58E-07	1,83E-07

Table 5.14: Area results (in square micrometers) for Cout function of a full adder.

	CSP	**NCSP**	**BDD**	**OpBDD**	**LBBDD**
CoutFA	28,78	28,78	33,73	33,09	28,78

The best implementations for the Cout function of a full adder are LBBDD and NCSP. They are faster, present better results in terms of consumption, and are smaller than the other implementations. It is due to the fact that they present small transistor count and transistor stacks.

5.3 Results for NPN-class Logic Functions up to 5-input

In order to analyze the impact of different network implementations, 500 arbitrary logic functions extracted from the NPN-class up to 5-input were selected. The total number of logic functions from NPN-class up to 5-input is 616625. This amount

makes unfeasible the network generation for all this set. Figure 5.1 presents the delay results. Figure 5.2 shows the dynamic consumption for the cells.

As it is possible to see, the results demonstrate that in general NCSP and LBBDD are the best choice to implement logic cells.

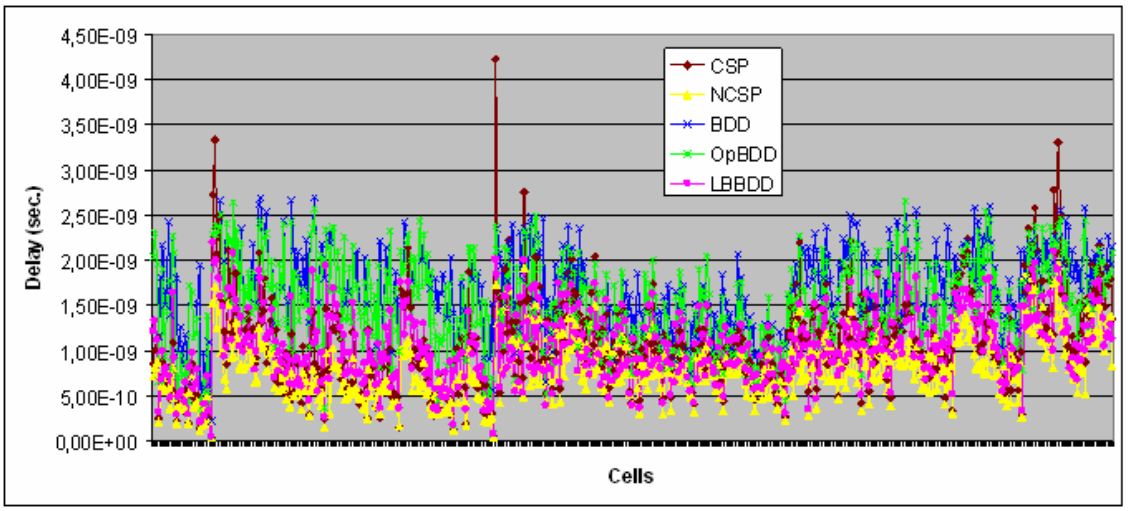


Figure 5.1: Delay results for 500 cells from NPN-class up to 5-input.

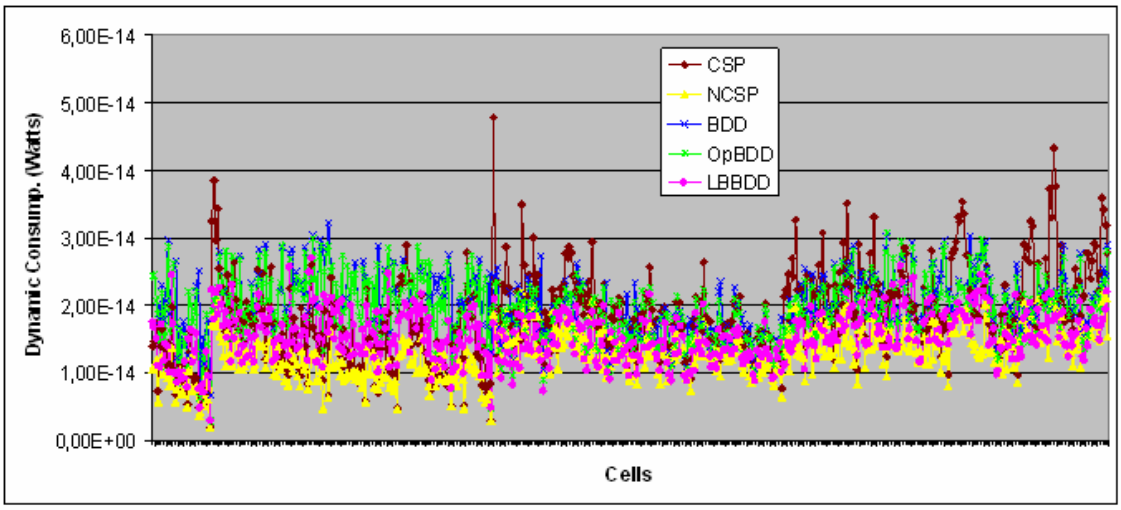


Figure 5.2: Dynamic consumption results for 500 cells from NPN-class up to 5-input.

Figure 5.3 presents the increase and decrease of transistor count for LBBDD, OpBDD and NCSP when comparing to CSP. NCSP is the logic style that implements a large number of cells without modifying the transistor count. LBBDD is capable of achieving the largest reduction of transistors between all logic styles.

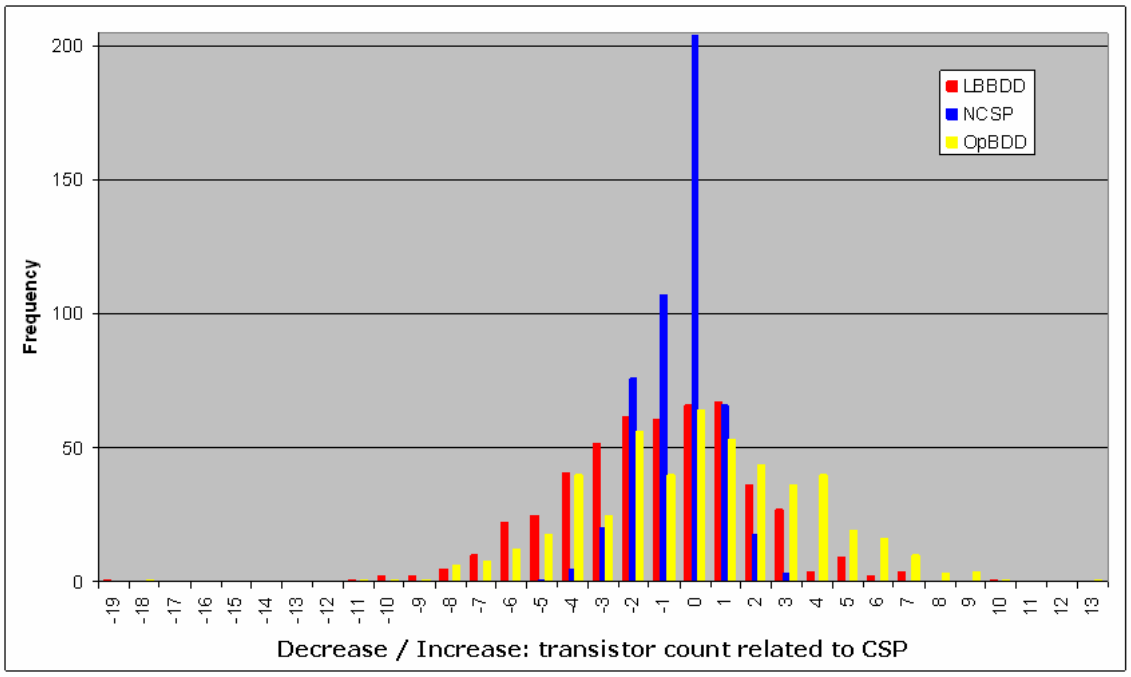


Figure 5.3: Increase and decrease in transistor count when comparing to CSP.

A subset of CSP networks that not respect the minimum number of transistors in series was selected from the total of 500 cells. For this obtained set of 423 cells, Figure 5.4 shows the worst achieved delay. Due to the large transistor stacks, CSP presents the worst results. LBBDD and NCSP can deliver more efficient networks. Figure 5.5 illustrates the average fanin for this subset of cells.

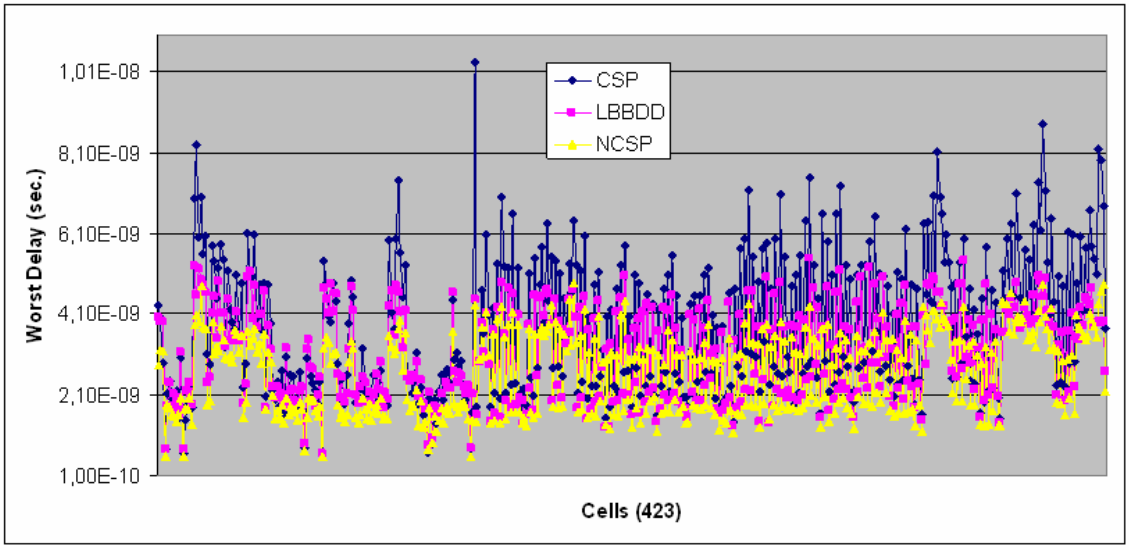


Figure 5.4: Worst delay for 423 cells that do not respect the minimum number of transistors in series when implemented in CSP.

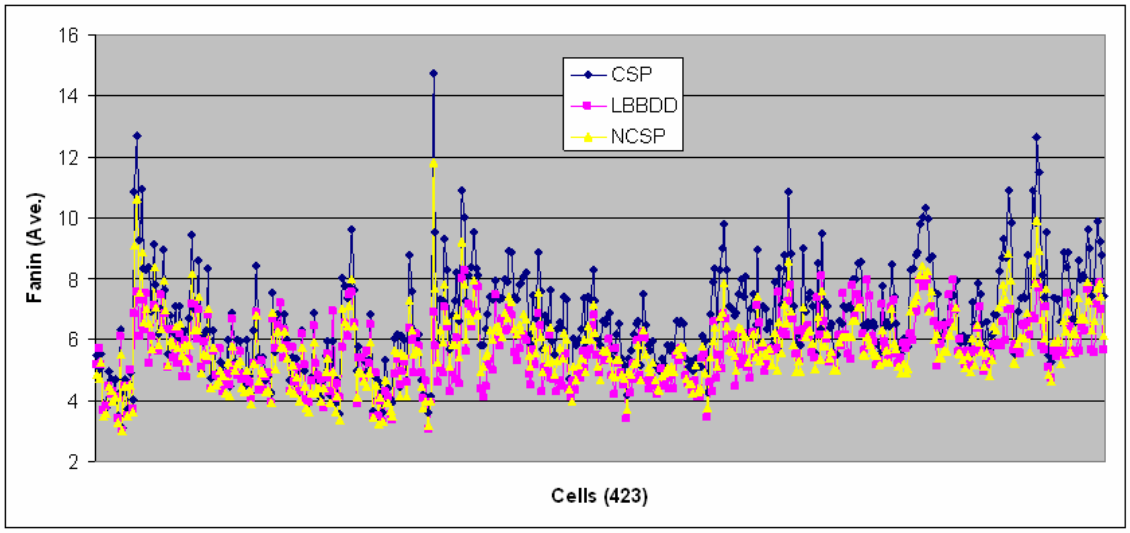


Figure 5.5: Average fanin for 423 cells that do not respect the minimum number of transistors in series when implemented in CSP.

From the total 500 cells implemented in LBBDD logic style, it was selected those that presented a bridge arrangement in at least one logic plane. For those cells, if the logic plane with a bridge arrangement presents a small transistor count than the complementary plane, it was generated a new plane using the duality property in order to minimize the total transistor count. Figure 5.6 presents the results obtained in this experiment for the 184 networks found. The blue line shows the reduction in transistor count, while the red line shows the increase in the transistor length. This increase can occur due to the fact that a plane is the dual of the other. When a given plane presents more than 4 parallel branches, the dual one derived from it will present transistor chains with more than 4 elements.

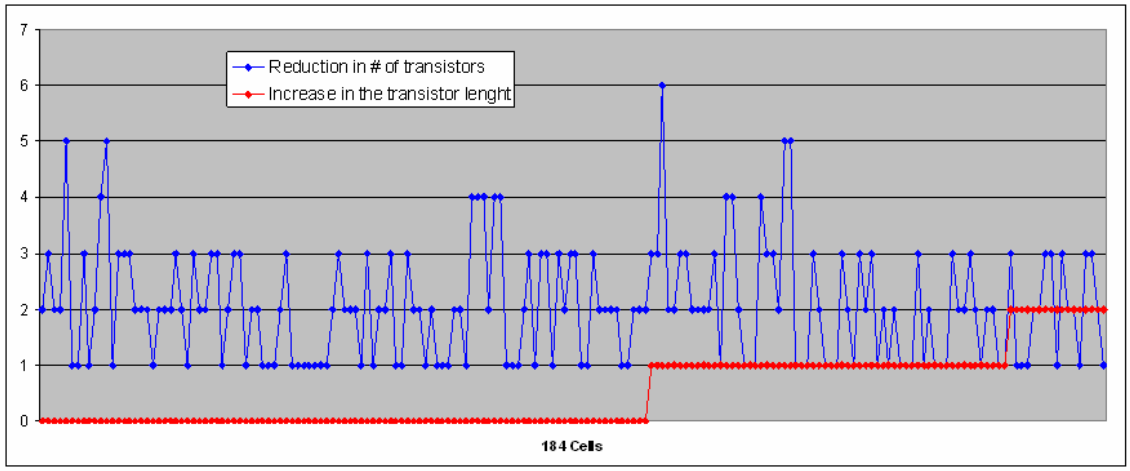


Figure 5.6: Experiment showing the reduction in transistor count and the increase in the transistor length when mixing LBBDD and Dual network generations.

5.4 Results for Logic Functions Unfeasible in CSP

Some logic functions extracted from (SCHNEIDER, 2007) were also used as benchmark functions in this thesis. These cells cannot be implemented in CSP logic since they present a huge number of transistor in series at least in one logic plane. Table 5.15 illustrates the transistor stacks for CSP, NCSP and LBBDD implementations.

NCSP and LBBDD logic styles respect the minimal number of transistor in series. This way, they permit to implement these logic functions as complex gates. On the other hand, CSP cannot be implemented in one single gate. These functions need to be decomposed in small ones in order to generate CSP transistor networks.

Table 5.16 shows the delay results for NCSP and LBBDD logic. Table 5.17 illustrates the power consumption. In Table 5.18, the leakage current is depicted.

Both logic styles presented similar results for the set of logic functions. In some cases LBBDD demonstrates a small advantage over NCSP. In others, NCSP shows a tiny gain. This experiment proves that it is important to analyze carefully at transistor level before choosing which complex transistor network will be used to compose the logic circuit.

Table 5.15: Number of transistor in series for functions from (SCHNEIDER, 2007).

	NCSP		**LBBDD**	
	PU	**PD**	**PU**	**PD**
F1	14	4	3	4
F2	15	4	3	4
F3	14	4	4	4
F4	15	4	4	4
F5	14	4	4	4
F6	16	3	3	4
F7	17	3	3	4

Table 5.16: Delay results (in seconds) for functions from (SCHNEIDER, 2007).

	NCSP	**LBBDD**
F1	1,12E-09	1,12E-09
F2	1,31E-09	1,34E-09
F3	1,09E-09	1,02E-09
F4	1,25E-09	9,89E-10
F5	8,15E-10	9,72E-10
F6	1,15E-09	1,15E-09
F7	9,53E-10	9,42E-10

Table 5.17: Power results (in Watts) for functions from (SCHNEIDER, 2007).

	NCSP	LBBDD
F1	2,32E-14	2,33E-14
F2	2,85E-14	2,86E-14
F3	3,36E-14	2,66E-14
F4	3,80E-14	2,69E-14
F5	2,10E-14	2,68E-14
F6	2,92E-14	2,92E-14
F7	2,67E-14	2,35E-14

Table 5.18: Leakage current (in Amperes) for functions from (SCHNEIDER, 2007).

	NCSP	LBBDD
F1	4,62E-07	4,41E-07
F2	4,99E-07	4,97E-07
F3	5,13E-07	5,29E-07
F4	5,16E-07	5,64E-07
F5	4,98E-07	5,34E-07
F6	5,52E-07	5,51E-07
F7	4,70E-07	5,18E-07

5.5 Branch-based vs. Factorized Functions

Another experiment was performed to investigate the impact over networks implemented using factorized and non-factorized forms. The set of logic functions extracted from Genlib 44-6 up to 4 inputs was used to implement CSP logic style. The networks were firstly implemented from the factorized form. A conversion from the factorized form to SOP representation was done. These SOP expressions were used to generate the second group of transistor networks.

Table 5.19 shows the delay results. Table 5.20 illustrates the power consumption. In table 5.21, the leakage is depicted. Finally, in Figure 5.22, the obtained areas are shown.

When the factorized form differs from the SOP form, the results point to a considerable gain for networks implemented from the optimized expression. This gain occurs in all cost axis: delay, power, and area.

Table 5.19: Average delay results (in seconds) for factorized and non-factorized forms.

	Factorized CSP	**Non-factorized CSP**
Cell0	3,53E-12	3,53E-12
Cell1	9,55E-11	1,68E-10
Cell2	1,64E-10	1,64E-10
Cell3	9,16E-11	1,33E-10
Cell4	5,89E-11	5,89E-11
Cell5	7,54E-11	7,54E-11
Cell6	1,80E-10	1,80E-10
Cell7	3,34E-10	3,34E-10
Cell8	1,79E-11	1,79E-11
Cell9	2,82E-11	2,82E-11
Cell10	3,76E-11	7,46E-11
Cell11	6,08E-11	6,08E-11
Cell12	4,10E-11	4,10E-11
Cell13	1,15E-10	1,15E-10
Cell14	6,65E-11	1,20E-10
Cell15	7,08E-11	2,06E-10
Cell16	7,57E-11	4,52E-10

Table 5.20: Power consumption (in Watts) for factorized and non-factorized forms.

	Factorized CSP	**Non-factorized CSP**
Cell0	3,88E-16	3,88E-16
Cell1	4,84E-15	7,65E-15
Cell2	5,33E-15	5,33E-15
Cell3	2,88E-15	4,16E-15
Cell4	2,86E-15	2,86E-15
Cell5	3,54E-15	3,54E-15
Cell6	4,79E-15	4,79E-15
Cell7	4,62E-15	4,62E-15
Cell8	1,41E-15	1,41E-15
Cell9	1,29E-15	1,29E-15
Cell10	2,06E-15	3,90E-15
Cell11	3,03E-15	3,03E-15
Cell12	1,97E-15	1,97E-15
Cell13	2,69E-15	2,69E-15
Cell14	3,02E-15	6,00E-15
Cell15	2,91E-15	7,60E-15
Cell16	3,42E-15	1,17E-14

Table 5.21: Leakage current (in Amperes) for factorized and non-factorized forms.

	Factorized CSP	**Non-factorized CSP**
Cell0	8,07E-08	8,07E-08
Cell1	1,24E-07	1,45E-07
Cell2	8,56E-08	8,56E-08
Cell3	1,70E-07	1,85E-07
Cell4	1,58E-07	1,58E-07
Cell5	1,77E-07	1,77E-07
Cell6	1,14E-07	1,14E-07
Cell7	6,92E-08	6,92E-08
Cell8	1,02E-07	1,02E-07
Cell9	9,62E-08	9,62E-08
Cell10	1,43E-07	1,64E-07
Cell11	9,69E-08	9,69E-08
Cell12	1,40E-07	1,40E-07
Cell13	8,57E-08	8,57E-08
Cell14	1,78E-07	1,56E-07
Cell15	1,56E-07	2,06E-07
Cell16	1,77E-07	2,23E-07

Table 5.22: Area (in micrometers) for factorized and non-factorized forms.

	Factorized CSP	**Non-factorized CSP**
Cell0	6,23	6,23
Cell1	8,15	8,15
Cell2	8,15	8,15
Cell3	11,67	13,59
Cell4	11,67	11,67
Cell5	11,67	11,67
Cell6	11,67	11,67
Cell7	14,39	17,10
Cell8	14,39	19,02
Cell9	13,59	24,46
Cell10	13,59	19,82
Cell11	13,59	13,59
Cell12	14,39	14,39
Cell13	14,39	17,10
Cell14	13,59	13,59
Cell15	13,59	13,59
Cell16	13,59	13,59

5.6 Fanin and Other Characteristics of P-class Logic Functions up to 4 Inputs

A comparative experiment to show how the topology of transistor networks influences the logical effort (fanin) of logic gates was also performed.

The set of evaluated functions include all the 3982 P-classes representing the set of non-constant 4-input logic functions. This set of functions was chosen because it contains simple functions that are more likely to be used in real designs as cells. For all the 3982 target functions, the network types described above were generated. Results are reported in Table 5.23. The data for each generation method are described in one column. For each method, the sum of the total number of transistors, length of longest transistor chain for pull-up (∑PU), length of the longest transistor chain for pull-down (∑PD), logical effort (average per cell input), number of functions that do not respect the lower bounds and the number of unfeasible functions is shown. The LBBDD is a clear winner for total number of transistors. This happens because even if some nodes are duplicated when generating the network, it is possible to remove several transistors, which compensates the duplication with advantages. LBBDD and NCSP respect the minimum number of transistor in series to implement the logic functions, so these methods have equal ∑PU lengths and ∑PD lengths, as shown in Table 5.23. However the total number of transistors is smaller for the LBBDD, which explains the advantage this method also has in terms of logical effort. Notice that the CSP respect the LB for the PU, as expected.

CSP, BDD and OpBDD methods produce functions not respecting the lower bounds. The BDD method is the one that produces the highest number of functions not respecting the minimum transistor stacks. However, all the functions it produces have at most 4 transistor in series (worst case path length for a BDD), and therefore they are considered feasible with a single cell. The only method to produce networks with more than 4 transistors in series is CSP. This is a result of using dual networks, which will result in excessive number of transistors in series when making a dual of a network that has many transistors in parallel. It is also observed that for networks with the same chain lengths, the one with a smaller transistor count is the winner.

Notice that these results show only the total values obtained in each logic style. Although they could point to the fact that the NCSP and LBBDD are strong candidates to generate optimized transistor networks, specific logic functions may present similar results when implemented in other logic styles.

Table 5.23: Comparison of different methods for P-class logic functions up to 4 variables.

	CSP	**NCSP**	**BDD**	**OpBDD**	**LBBDD**
∑ # transistors	75530	75889	76774	73438	72307
∑PU length	11954	11954	15538	14227	11954
∑PD length	17009	14242	15538	15321	14242
Aver. logical effort	4.54	3.83	4.35	4.07	3.68
#f not respecting LB	2312	0	3148	2373	0
# of unfeasible f	1546	0	0	0	0

5.7 Final Considerations

This chapter presented some experimental results with different sets of logic functions. These sets were implemented in different logic styles and were compared to demonstrate that depending on the target logic function there is a possibility of achieving a better implementation in terms of delay, power or area.

The set of logic functions used to perform the experiments of this chapter are described in Appendix D of this thesis.

6 CONCLUSIONS AND FUTURE WORKS

This work presented an automated flow for generating and evaluating transistor cell networks. The main goal of the work proposed herein was to develop an approach able to generate logic cell networks on-the-fly, considering different logic styles, and evaluate these networks using estimative techniques.

In a first moment, a review about switch theory was done. A switch network classification was compiled in order to clarify the switch network properties and to present the richness of the switch theory. Also, we proposed a factorization method to optimized Boolean expressions. These optimized expressions are suitable to be used as input to implement efficient transistor networks, as it was presented in the experimental results.

Several generation methods for transistor networks were presented, from the traditional CSP logic style to the new NCSP proposed by Schneider (2007), which is a network solution that achieves the minimum length for transistor chains needed to implement a given logic function. A review on graph-based networks implementation was done, discussing the most relevant researches on this field. In the sequence, a new static and disjoint logic style was proposed. This logic, called LBBDD, is a BDD-based solution and demonstrated to be very promising. Like NCSP, this logic style delivers networks with minimum number of transistors in series in pull-up and pull-down planes. The advantage over the NCSP is that factorization is not necessary to achieve optimized networks, since the optimizations are performed in the BDD structure. Also, it is capable of delivering Wheatstone-bridge networks. This kind of network tends to be more efficient than traditional series-parallel arrangements as it minimizes the total transistor count. Clearly, the method proposed herein is more general as it can generate all the (logically complementary) categories in the switch network classification we proposed.

To evaluate the generated transistor networks, some estimation techniques were employed. To investigate the delay of different implementations of same logic functions, the Elmore delay model was implemented. Although this delay model is not an accurate solution, it is an excellent approach to perform a first-order timing estimation that delivers good information about the delay behavior of logic cells. To

investigate the dynamic power consumption, a method proposed by Chiappetta (2008) was implemented. This method is based on the intrinsic capacitance computation. Thus, the short-circuit power dissipation was not considered in this work. To estimate the leakage current of the generated networks, three models were implemented. The first one is a simple solution to compute subthreshold leakage only. The second one is a straightforward approach to compute gate leakage only. Finally, the third model is an accurate method to estimate subthreshold and gate leakage together. In the experimental results, the third one was used. However, the other two can be applied when the designer needs to evaluate only a unique leakage component, disregarding the others. Finally, to evaluate area we proposed a naïve approach which considers some technology process distances, extracted from the technology process datasheet, to compute and estimate the cell width. The cell height has not been investigated in the current version. So, the height is fixed in a given value in order to achieve the area results. Notice that this approach is useful when thinking in standard-cells design flow, since all cells present the same height. Considering a height value that is sufficient to perform the internal routing of all cells, we can compare the implementations with a good exactness as demonstrated in the validation experiment.

The results show that LBBDD and NCSP in general are good alternatives to implement logic functions. However, they are not the best choice for every logic function. For instance, there is no need to use this kind of network generation when the target logic function is unate and do not present repeated literals. In this case, CSP networks are able to attend the minimum implementation in terms of transistor in series and total count. On the other hand, XOR3 and XOR4 become feasible in LBBDD and NCSP. The results demonstrated that, for the used set of logic functions, LBBDD also presents a significant reduction in terms of area if compared to the other logic styles. This happens because several transistors are removed from the network during the generation process. Mux-Based networks achieved the worst area results, showing that a non-factorized form can negatively impact on the generated cell network.

Two CAD tools, presented in Appendix E, were implemented during this work. They contain several methods and ideas discussed herein. These tools can be used to help designers to generate and investigate logic cells behavior. As future work it is intended to implement a technology mapping tool capable of using the networks and information generated by the methods presented in this thesis.

REFERENCES

ALIOTO, M. A Simple and Accurate Model of Input Capacitance for Power Estimation in CMOS Logic. In: IEEE INTERNATIONAL CONFERENCE ON ELECTRONICS, CIRCUITS AND SYSTEMS, ICECS, 14., 2007, Marrakech, Marrocos. **Proceedings**… New York, USA: IEEE, 2007. p. 431-434.

AVCI, M.; YILDIRIM, T. General Design Method for Complementary Pass Transistor Logic Circuits. **Electronics Letters**, [S.l.], v. 39, pt. 1, p. 46-48, Jan. 2003

BERKELAAR, M.; JESS, J. Technology Mapping for Standard-cell Generators. In: INTERNATIONAL CONFERENCE COMPUTER-AIDED DESIGN, ICCAD, 1988, Santa Clara, USA. **Digest of Technical Papers**. New York, USA: IEEE, 1988. p. 470-473.

BERTACCO, V. et al. **Decision Diagrams and Pass Transistor Logic Synthesis**. [S.l.]: Stanford University, 1997. (Technical Report n. CSL-TR-97-748).

BHATTACHARYA, D. et al. Design Optimization with Automated Flex-cell Creation. In: KEUTZER, K.W. (Ed.). **Closing the Gap Between ASIC & Custom**: tools and techniques for high-performance ASIC design. Boston: Kluwer Academic, 2002. p. 14-23.

BOGLIOLO, A. et al. Gate-Level Power and Current Simulation of CMOS Integrated Circuits. **IEEE Transactions on VLSI Systems**, New York, USA, v. 5, n. 4, p. 473-488, Dec. 1997.

BOLLIG, B.; WEGENER, I. Improving the Variable Ordering of OBDDs Is NP-Complete. **IEEE Transactions on Computers**, New York, USA, v. 45, n. 9, p. 993-1002, Sept. 1996.

BRAYTON, R. Factoring logic functions. **IBM Journal of Research and Development**, Riverton, USA, v. 31, n. 2, p. 187-198, Mar. 1987.

BRYANT, R. E. Graph-based algorithms for Boolean function manipulation. **IEEE Transactions on Computers**, New York, USA, v.35, n.8, p. 677-691, Aug. 1986.

BUDDY. Available at: <http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/index.html>. Visited on: June 2008.

BUTZEN, P.F. et al. Subthreshold and Gate Leakage Estimation in Complex Gates. In: IEEE INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS, IWLS, 17., 2008, Lake Tahoe, USA. **Proceedings…** NewYork, USA: IEEE, 2008.

CARDOSO, T. M. G. et al. Speed-up of ASICs Derived from FPGAs by Transistor Network Synthesis Including Reordering. In: INTERNATIONAL SYMPOSIUM ON QUALITY ELECTRONIC DESIGN, ISQED, 9., 2008, San Jose, USA. **Proceedings**…New York, USA: IEEE, 2008. p. 47-52.

CARLSON, B. S.; CHEN, C. Y. R. Effects of transistor reordering on the performance of MOS digital circuits. In: MIDWEST SYMPOSIUM ON CIRCUITS AND SYSTEMS, 1992. **Proceedings**… [S.l.:s.n.], 1992. v.1, p. 121–124.

CHEN, W.-K. **The VLSI Handbook**. Boca Raton, USA: CRC Press, 2000.

CHENG, Z. et al. Estimation of Standby Leakage Power in CMOS Circuits Considering Accurate Modeling of Transistor Stacks. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 4., 1998, Monterey, USA. **Proceedings**... [S.l.: s.n.], 1998. p. 239-244.

CHIAPPETTA, E.J.D et al. A Simple Model to Estimate Intrinsic Power Consumption in CMOS Logic Gates. In: STUDENT FORUM ON MICROELECTRONICS, SForum, 8., 2008, Gramado, Brasil. **Proceedings**... Porto Alegre, Brasil: SBC, 2008.

CORREIA, V.P.; REIS, A.I. Classifying n-Input Boolean Functions. IBERCHIP WORKSHOP, IWS, 7., 2001, Montevideo, Uruguai. **Proceedings**… [S.l.: s.n.], 2001. p. 58.

CORREIA, V.; REIS, A. Advanced technology mapping for standard-cell generators. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 17., 2004, Porto de Galinhas, Brasil. **Proceedings**…Los Alamitos: IEEE, 2004. p. 254-259.

CUDD. Available at: <http://vlsi.colorado.edu/~fabio/CUDD/>. Visited on: June 2008.

DEMICHELI, G. **Synthesis and Optimization of Digital Circuits**. New York, USA: McGraw-Hill, 1994.

DRECHSLER, R.; BECKER, B. **Binary Decision Diagrams**: Theory and Implementation. Boston, USA: Kluwer Academic, 1998.

DROZDEL, A. **Estrutura de Dados e Algoritmos em C++**. São Paulo, Brasil: Thomson Learning, 2002.

DUFFIN, R. J. Topology of series-parallel networks. **Journal of Mathematical Analysis and Applications**, San Diego, USA, v. 10, p. 303-318, 1965.

EBENDT, R.; FEY, G.; DRECHSLER, R. **Advanced BDD Optimization**. Dordrecht, Netherlands: Springer, 2005.

ELMORE, W. The transient response of damped linear networks with particular regard to wideband amplifiers. **Journal Applied Physics**, Woodbury, USA, v. 19, n. 1, p. 55-63, Jan. 1948.

ERCEGOVAC, M.; LANG, T.; MORENO, J.H. **Introdução aos Sistemas Digitais**. Porto Alegre, Brasil: Bookman, 2000. p. 229-234.

EVEN, S. **Graph Algorithms**. London, England: Pitman Publishing, 1979.

FISHER, C. et al. Optimization of Standard Cell Libraries for Low Power, High Speed, or Minimal Area Designs. In: IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE, CICC, 17., 1996, San Diego, USA. **Proceedings**... New York, USA: IEEE, 1996. p. 493-496, 1996.

GAVRILOV, S.; GLEBOV, A. BDD Based Circuit Level Structural Optimization for Digital CMOS. In: INTERNATIONAL WORKSHOP MULTI-ARCHITECTURE LOW POWER DESIGN, 1., 1999, Moscou, Rússia. **Proceedings**... [S.l.:s.n.], 1999. 45-49.

GU, R. X.; ELMASRY, M. I. Power Distribution Analysis and Optimization of Deep Submicron CMOS Digital Circuit. **IEEE Journal of Solid-State Circuits**, New York, v.31, n.5, p.707-713, May 1996.

HARARY, F. **Graph Theory**. [S.l.]: Perseus Books Group, 1994.

HARRISSON, M.A. **Introduction to switching and automata theory**. [S.l.]: McGraw-Hill, 1965.

HENTSCHKE, R. **Blue Macaw Didactic Placement v0.8b**. Available at: < http://www.inf.ufrgs.br/~renato/bluemacaw/index.html>. Visited on: June 2008.

ISAEVA, T. Switch-Level BDD-based Synthesis Algorithm. In: INTERNATIONAL WORKSHOP MULTI-ARCHITECTURE LOW POWER DESIGN, 1., 1999, Moscou, Rússia. **Proceedings**... [S.l.:s.n.], 1999. p. 39-44.

JIANG, Y.; SAPATNEKAR, S.; BAMJI, C. Technology mapping for high-performance static CMOS and pass transistor logic designs. **IEEE Transactions on VLSI Systems**, New York, USA, v. 9, n. 5, p. 577-589, Oct. 2001.

KAGARIS, D.; HANIOTAKIS, T. Transistor Level Optimization of Supergates. In: INTERNATIONAL SYMPOSIUM ON QUALITY ELECTRONIC DESIGN, ISQED, 7., 2006, Santa Clara, USA. **Proceedings**... New York, USA: IEEE, 2006. p. 1-10.

KAGARIS, D.; HANIOTAKIS, T. A Methodology for Transistor-Efficient Supergate Design. **IEEE Transactions on VLSI Systems**, New York, USA, v.15, n.4, p. 488-492, Apr. 2007.

KARMA. Available at: <http://www.inf.ufrgs.br/nangate/> Visited on: June 2008.

KARNAUGH, M. The Map Method for the Synthesis of Combinational Circuits. **AIEE Transactions**: Communications and Electronics, [S.l.], v. 72, pt. 1, p. 593-599, Nov. 1953.

KEUTZER, K. Dagon: Technology binding and local optimization by DAG matching. In: DESIGN AUTOMATION CONFERENCE, DAC, 24., 1986, Miami, USA. **Proceedings**… [S.l.:s.n.], 1987. p. 341-347.

KEUTZER, K.; RICHARDS, D. Computational complexity of logic synthesis and optimization. In: IEEE INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS, IWLS, 2., 1989. **Proccedings**… [S.l.:s.n.], 1989. p. 1-15.

KUKIMOTO, Y.; BRAYTON, R.; SAWKAR, P. Delay-optimal technology mapping by DAG covering. In: DESIGN AUTOMATION CONFERENCE, DAC, 35., 1998, San Francisco, USA. **Proceedings**… [S.l.]: ACM, 1998. p. 348-351.

LAI, Y.; JIANG Y.; CHU, H. BDD Decomposition for Mixed CMOS/PTL Logic Circuit Synthesis. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2005, Kobe, Japan. **Proceedings**… [S.l.:s.n.], 2005. p. 5649-5652.

LAWLER, E.L. An approach to multilevel Boolean minimization. **Journal of the ACM**, New York, USA, v.11, n.3, p.283-295, July 1964.

LEE, C. Y. Representation of Switching Circuits by Binary-Decision Programs. **Bell Systems Technical Journal**, [S.l.], n.38, p.985-999, 1959.

LIU, D.; SVENSSON, C. Power Consumption Estimation in CMOS VLSI Chips. **IEEE Journal of Solid State Circuits**, New York, USA, v. 29, n.6, p. 663-670, June 1994.

LIU, C. R.; ABRAHAM, J. A. Transistor Level Synthesis for Static CMOS Combinational Circuits. In: GREAT LAKES SYMPOSIUM ON VLSI, GLSVLSI, 9., 1999, Ann Arbor, USA. **Proceedings**… New York, USA: IEEE, 1999. p. 172-175.

MA, S.; FRANZON P. Energy Control and Accurate Delay Estimation in the Design of CMOS Buffers. **IEEE Journal of Solid-State Circuits**, New York, USA, v.29, n.9, p. 1150-1153, Sept. 1994.

MARQUES, F.S.; ROSA, L.S.; RIBAS, R.P.; SAPATNEKAR, S.S.; REIS, A.I. DAG Based Library-Free Technology Mapping. In: ACM GREAT LAKES SYMPOSIUM ON VLSI, GLSVLSI, 17., 2007, Lago Maggiore, Italy. **Proceedings** …[S.l.]: ACM, 2007.

MCCLUSKEY, E. J. Minimization of Boolean Functions. **Bell Systems Technical Journal**, [S.l.], v. 35, p. 1417-1444, June 1956.

MCGEER, P. et al. ESPRESSO-SIGNATURE: A new exact minimizer for logic functions. In: DESIGN AUTOMATION CONFERENCE, DAC, 30., 1993, Dallas, USA. **Proccedings**… [S.l.]: ACM, 1993.

MISCHENKO, A. et al. **Technology mapping with Boolean matching, supergates and choices**. Berkeley: EECS Dept., UC Berkeley, 2005. (ERL Technical Report).

MINTZ, A.; GOLUMBIC, M.C. Factoring Boolean functions using graph partitioning. **Discrete Applied Mathematics**, [S.l.], n.149, p.131-135, May 2005.

MOORE, E.F. Table of four-relay contact networks. In: HIGONNET, R.A.; GREA, R.A. (Ed.). **Logical Design of Electrical Circuits**. [S.l.]: McGraw-Hill, 1958.

NAJM, F. A survey of power estimation techniques in VLSI circuits. **IEEE Transactions on VLSI Systems**, New York, USA, v. 2, p. 446-455, Dec. 1994.

NANGATE. Available at: < http://www.nangate.com>. Visited on: June 2008.

NÈVE, A.; FLANDRE, D. Branch-Based Logic for High Performance Carry-Select Adders in 0.25 µm Bulk and Silicon-On-Insulator CMOS Technologies. In: INTERNATIONAL WORKSHOP-POWER AND TIMING MODELING, OPTIMIZATION AND SIMULATION, PATMOS, 11., 2001, Yverdon-Les-Bains, Switzerland. **Proceedings**… [S.l.]: IEEE, 2001.

PANDA, R. et al. Migration: A new technique to improve synthesized designs through incremental customization. In: DESIGN AUTOMATION CONFERENCE, DAC, 35., 1998, San Francisco, USA. **Proceedings**… [S.l.]: ACM, 1998. p. 388-391.

PARK, J. C.; MOONEY III, V. J. Sleepy Stack Leakage Reduction. **IEEE Transactions on VLSI Systems**, New York, USA, v.14, n.11, p.1250-1262, Nov. 2006.

PIGUET, C. et al. A Metal-Oriented Layout Structure for CMOS Logic. **IEEE Journal of Solid-state Circuits**, New York, USA, v. 19, n.3, p. 425- 436, June 1984.

PIGUET, C. et al. Low-power low-voltage digital CMOS cell design. In: INTERNATIONAL WORKSHOP-POWER AND TIMING MODELING, OPTIMIZATION AND SIMULATION, PATMOS, 4., 1994, Barcelona, Spain. **Proceedings**… [S.l.:s.n.], 1994. p. 132–139.

PIGUET, C. et al. Low-power low-voltage standard cell libraries. In: EUROPEAN SOLID-STATE CIRCUITS CONFERENCE, ESSCIRC, 21., Lille, France. **Proceedings**… [S.l.:s.n.], 1995.

POLI, R. E. B.; RIBAS R. P.; REIS A. I. Unified Theory to Build Cell-Level Transistor Networks from BDDs. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 16., 2003, São Paulo, Brasil. **Proceedings**… Los Alamitos: IEEE, 2003. p. 199–204.

PRUNTY, C. et al. Optimum Tapered Buffer. **IEEE Journal of Solid-State Circuits**, New York, USA, v. 27, n.1, p. 118-119, Jan. 1992.

QUINE, W. V. A Way To Simplify Truth Functions. **American Mathematical Monthly**, Washington, v. 62, p. 627-631, 1955.

RABAEY, J.M.; CHANDRAKASAN. A.; NIKOLIC, B. **Digital Integrated Circuits**: A Design Perspective. 2nd ed. Upper Saddle River: Prentice Hall, 2005.

RAO, R. M. et al. Efficient Technique for Gate Leakage Estimation. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 9., 2003, Seoul, Korea. **Proceedings**… [S.l.]: ACM, 2003. p. 100-103.

REIS, A. I.; ROBERT, M.; AUVERGNE, D.; REIS R. Associating CMOS Transistors with BDD Arcs for Technology Mapping. **Electronics Letters**, [S.l.], v. 31, n. 14, p. 1118–1120.

REIS, A.I. **Assignation Technologique sur Bibliotheques Virtuelles de Portes Complexes CMOS**. 1998. 123 f. These (Doctorate m Electronique, Optronique et Systemes) - Université de Montpellier, Montpellier, France.

ROADMAP, International Technology Roadmap for Semiconductors. 2004. Available at: <http://public.itrs.net>. Visited on: June 2008.

ROSA, L.S.; RIBAS, R.P.; REIS, A.I. Fast Transistor Networks from BDDs. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 19., 2006, Ouro Preto, Brasil. **Proceedings**… New York: ACM, 2006. p. 137-142

ROSA, L.; RIBAS, R.; REIS, A. A Comparative Study of CMOS Gates with Minimum Transistor Stacks. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 20., 2007, Rio de Janeiro, Brasil. **Proceedings**… New York: ACM, 2007. p. 93-98

ROSSELLÓ, J. L.; SEGURA, J. Accurate modeling of leakage currents in nanometer CMOS technologies. **Electronics Letters**, [S.l.], v. 41, n. 3, p. 122-123, 2005.

ROY, K. et al. Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits. **Proceedings of the IEEE**, New York, v. 91, n. 2, p. 302-327, Feb. 2003.

ROY, R.; BHATTACHARYA, D.; BOPPANA, V. Transistor-level optimization of digital designs with flex cells. **IEEE Transactions on Computers**, New York, v. 38, n. 2, p. 53-61, Feb. 2005.

RUDELL, R. L. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 1993, Santa Clara. **Digest of Technical Papers**. New York, USA: IEEE, 1993. p. 42-47.

RUPESH, S.S. **Synthesis for Nanometer Technologies**. 2004. Thesis. University of Minnesota, Minnesota, USA.

SASAO, T. **Switching Theory for Logic Synthesis**. Boston: Kluwer Academic, 2000.

SCHNEIDER, F.R. **Explorando técnicas para estimativa de desempenho em portas lógicas CMOS**. 2004. Graduate Degree Thesis. Engenharia de Computação.Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil.

SCHNEIDER, F.R.; RIBAS, R.P.; REIS, A.I. Fast CMOS Logic Style Using Minimum Transistor Stack for Pull-up and Pull-down Networks. In: IEEE INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS, IWLS, Vail, USA, 15., 2006. **Proceedings**... [S.l.:s.n.], 2006. p. 134-141.

SCHNEIDER, F.R. **Building Transistor-Level Networks Following the Lower Bound on the Number of Stacked Switches**. 2007. Master Deegre Thesis. Mestrado em Ciência da Computação. Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil.

SCOTT, K.; KEUTZER, K. Improving Cell Libraries for Synthesis. In: IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE, CICC, 1994, San Diego, USA. **Proceedings**... New York: IEEE, 1994. p. 128-131.

SECHEN, C.; GUAN, B. Large standard cell libraries and their impact on layout area and circuit performance. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, ICCD, 1996, Austin, USA. **Proceedings**… New York: IEEE, 1996. p. 378-383.

SECHEN, C. Libraries: lifejacket or straitjacket. In: DESIGN AUTOMATION CONFERENCE, DAC, 40., 2003, Anaheim, USA. **Proceedings**... New York: ACM, 2003. p. 642-643.

SENTOVICH, E. et al. **SIS**: A system for sequential circuit synthesis. Berkeley: EECS Department, University of California, 1992. (Technical Report n. UCB/ERL M92/41).

SHANNON, C.E. A symbolic Analysis of Relay and Switching Circuits. **Transactions American Institute of Electrical Engineers**, New York, v.57, p. 38-80, May 1938.

SHANNON, C.E. **Realization of All 16 Switching Functions of Two Variables Requires 18 Contacts**. [S.l.]: Bell Laboratories, 1953a.

SHANNON, C.E. **Machine Aid for Switching Circuit Design**. [S.l.]: Bell Laboratories, 1953b.

STOK, L.; LYER, M.A.; SULLIVAN, A.J. Wavefront Technology Mapping. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 1999, Munich, Germany. **Proceedings**... New York: IEEE, 1999. p. 531–536.

SUTHERLAND, I.; SPROULL, B.; HARRIS, D. **Logical Effort:** Designing Fast CMOS Circuits. San Francisco, USA: Morgan Kaufmann, 1999.

THORP, T.J.; YEE, G.S.; SECHEN, C.M. Design and synthesis of dynamic circuits. **IEEE Transactions on VLSI Systems**, New York, v. 11, n. 1, p. 141-149, Feb. 2003.

TSUI, C. Y.; PEDRAM, M.; DESPAIN A. Efficient estimation of dynamic power dissipation under a real delay model. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 1993, Santa Clara. **Digest of Technical Papers**. New York, USA: IEEE, 1993. p. 224-228.

UEHARA, T.; VANCLEEMPUT, W. M. Optimal Layout of CMOS Functional Arrays. **IEEE Transactions on Computers**, Los Alamitos, v.c-30, n. 5, p. 305-312, May 1981.

UYEMURA, J. P. **CMOS Logic Circuit Design**. Boston: Kluwer Academic, 1999.

VEENDRICK, H.J.M. Short-Circuit Dissipation of Static CMOS Circuitry and its Impact on the Design of Buffer Circuits. **IEEE Journal of Solid State Circuits**, New York, v.SC-19, n.4, p. 468-473, Aug. 1984.

VEMURA, S.R. et al. Variable-taper CMOS buffer. **IEEE Journal of Solid-State Circuits**, New York, v.26, n.9, p. 1265-1269, Sept. 1991.

VUJKOVIC, M.; SECHEN, C. Optimized power-delay curve generation for standard cell ICs. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 2002, San Jose. **Digest of Technical Papers**. New York, USA: ACM, 2002.

WAGNER, F.; RIBAS, R.; REIS, A. **Fundamentos de Circuitos Digitais**. Porto Alegre: Sagra Luzzatto, 2006.

WESTE, N.H.E.; HARRIS, D. **CMOS VLSI Design**: A Circuits and Systems Perspective. 3rd ed. Boston: Pearson/Addison Wesley, 2005.

YANG, S. et al. Accurate Stacking Effect Macro-modeling of Leakage Power in Sub-100nm Circuits. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 18., 2005, Kolkata, India. **Proceedings**... New York: IEEE, 2005. p. 165-170.

YOSHIDA, H. et al. A Structural Approach for Transistor Circuit Synthesis. **IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences**, [S.l.], v.E89-A, n. 12, Dec. 2006.

ZHAO, W.; CAO, Y. New generation of Predictive Technology Model for sub-45nm early design exploration. **IEEE Transactions on Electron Devices**, New York, v. 53, n. 11, p. 2816-2823, Nov. 2006.

ZIESEMER JUNIOR, A.M.; LAZZARI, C.; REIS, R. Automatic Transistor-Level Layout Generator of CMOS Cells. In: SOUTH SYMPOSIUM ON MICROELECTRONICS, SIM, 22., 2007, Porto Alegre, Brasil. **Proceedings**... Porto Alegre, Brasil: SBC, 2007. p.57-60.

APPENDIX A AN ACADEMIC LIBRARY DESCRIPTION

This library description corresponds to a subset of the *lib2.genlib* that is an academic library distribute with the SIS technology mapping tool (SENTOVICH, 1992).

```
GATE inv1  928.00 O=!a;
  PIN a INV 0.0514 999.0 0.4200 4.7100 0.4200 3.6000
GATE nand2  1392.00 O=!(a*b);
  PIN a INV 0.0777 999.0 0.6400 4.0900 0.4000 2.5700
  PIN b INV 0.0716 999.0 0.4600 4.1000 0.3700 2.5700
GATE nand4  2320.00 O=!(a*b*c*d);
  PIN a INV 0.1030 999.0 1.2700 3.6200 0.6700 2.3900
  PIN b INV 0.0980 999.0 1.0900 3.6100 0.6100 2.3900
  PIN c INV 0.0980 999.0 0.8200 3.6200 0.5500 2.4000
  PIN d INV 0.1050 999.0 0.5800 3.6200 0.3800 2.3900
GATE nor2  1392.00 O=!(a+b);
  PIN a INV 0.0736 999.0 0.3300 3.6400 0.4500 3.6400
  PIN b INV 0.0968 999.0 0.5000 3.6400 0.7000 3.6600
GATE nor4  2320.00 O=!(a+b+c+d);
  PIN a INV 0.0887 999.0 0.4100 5.9100 1.1600 3.2000
  PIN b INV 0.0867 999.0 0.8500 5.9100 1.5300 3.1800
  PIN c INV 0.0867 999.0 1.1100 5.9200 1.7500 3.1900
  PIN d INV 0.0887 999.0 1.2700 5.9100 1.9400 3.2000
GATE aoi21  1856.00 O=!((a*b)+c);
  PIN a INV 0.1029 999.0 0.7500 3.5200 0.6700 2.5300
  PIN b INV 0.0908 999.0 0.6700 3.6400 0.6200 2.5200
  PIN c INV 0.1110 999.0 0.5800 3.6400 0.2100 1.2800
GATE aoi22  2320.00 O=!((a*b)+(c*d));
  PIN a INV 0.1019 999.0 0.9200 3.4600 0.9400 2.7900
  PIN b INV 0.0908 999.0 0.8400 3.6400 0.8500 2.7900
  PIN c INV 0.0958 999.0 0.6100 3.6400 0.4900 2.9300
  PIN d INV 0.0988 999.0 0.7000 3.6400 0.5400 2.9300
```

Figure A: A subset of the *lib2.genlib* academic library.

A cell is specified in the following format:

GATE <*cell_name*> <*cell_area*> <*cell_logic_function*>

PIN <*pin_name*> <*phase*> <*input_load*> <*max_load*>
<*rise_block_delay*> <*rise_fanout_delay*>
<*fall_block_delay*> <*fall_fanout_delay*>

<*cell_name*> is the name of the cell in the cell library.

<*cell_area*> defines the relative area cost of the cell. It is a floating point number, and may be in any unit system convenient for the user.

<*cell_logic_function*> is an equation written in conventional algebraic notation using the operators "+" for OR, "*" or nothing (space) for AND, "!" or "'" (post-fixed) for NOT, and parentheses for grouping. The names of the literals in the equation define the input pin names for the cell; the name on the left hand side of the equation defines the output of the cell. The equation terminates with a semicolon.

<*pin_name*> must be the name of a pin in the <*cell_logic_function*>, or it * to specify identical timing information for all pins.

<*phase*> is INV, NONINV, or UNKNOWN corresponding to whether the logic function in negative unite, positive unate, or binate in this variable respectively. This is required for the separate rise-fall delay model.

<*input_load*> gives the input load of this pin. It is a floating point value, in arbitrary units convenient for the user.

<*max_load*> specifies a loading constraint for the cell. It is a floating point value specifying the maximum load allowed on the output.

<*rise_block_delay*> and <*rise_fanout_delay*> are the rise-time parameters for the timing model. They are floating point values, typically in the units nanoseconds, and nanoseconds/unit_load respectively.

<*fall_block_delay*> and <*fall_fanout_delay*> are the fall-time parameters for the timing model. They are floating point values, typically in the units nanoseconds, and nanoseconds/unit_load respectively.

The delay information for the most critical pin is used to determine the delay for the logic gate.

APPENDIX B XOR TRANSISTOR SCHEMATICS

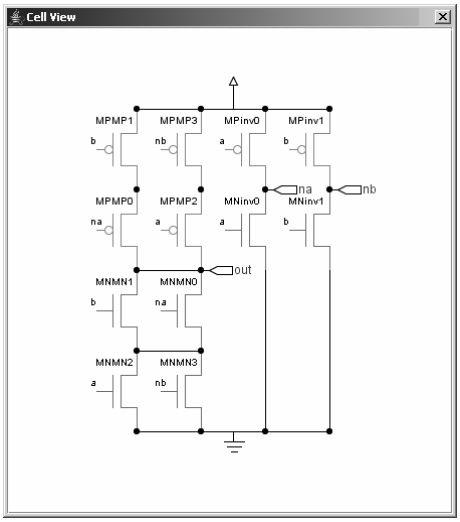


Figure B1: XOR2 in CSP, NCSP, BDD, OpBDD and LBBDD logic styles.

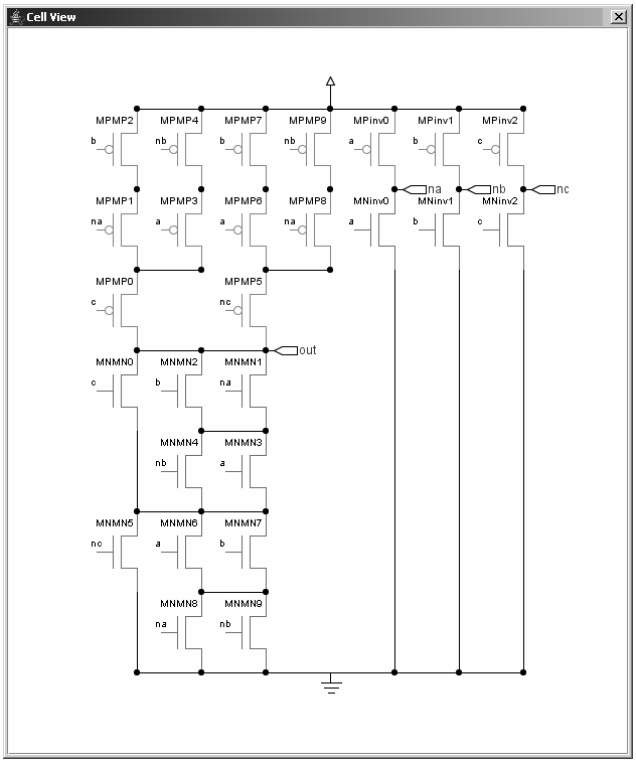


Figure B2: XOR3 in CSP logic style.

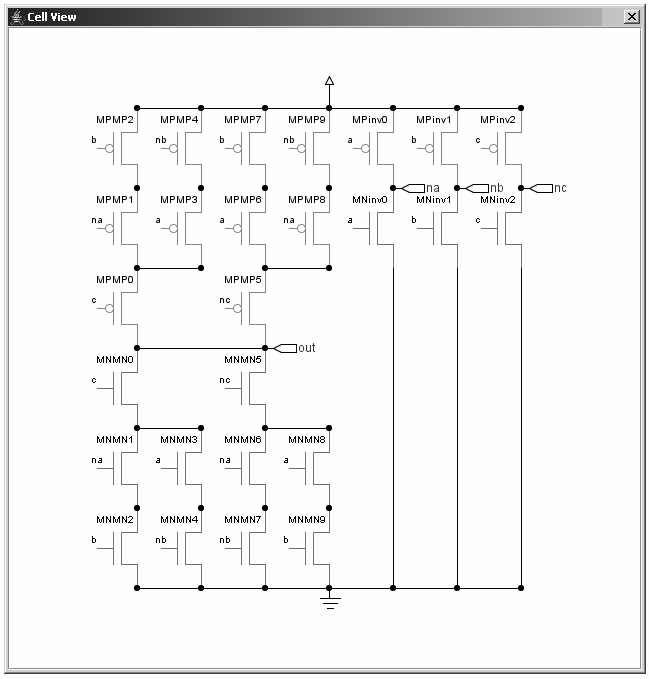


Figure B4: XOR3 in NCSP logic style.

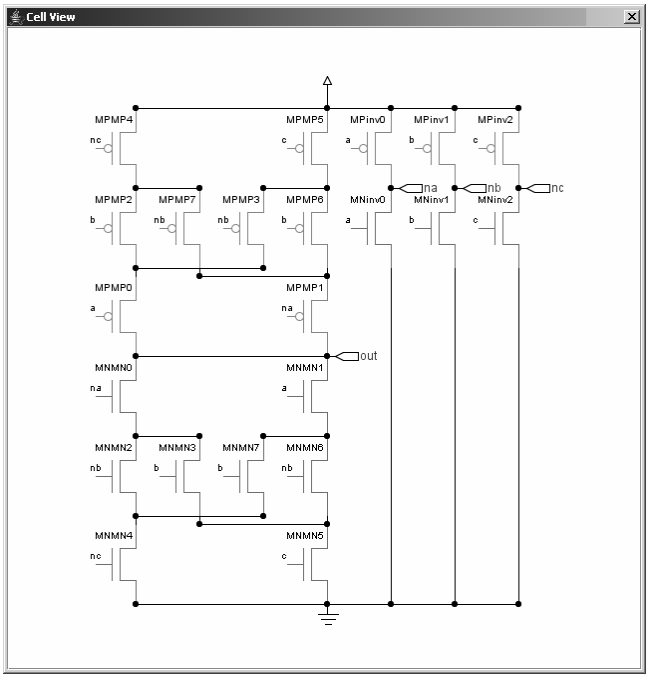


Figure B5: XOR3 in BDD, OpBDD and LBBDD logic styles.

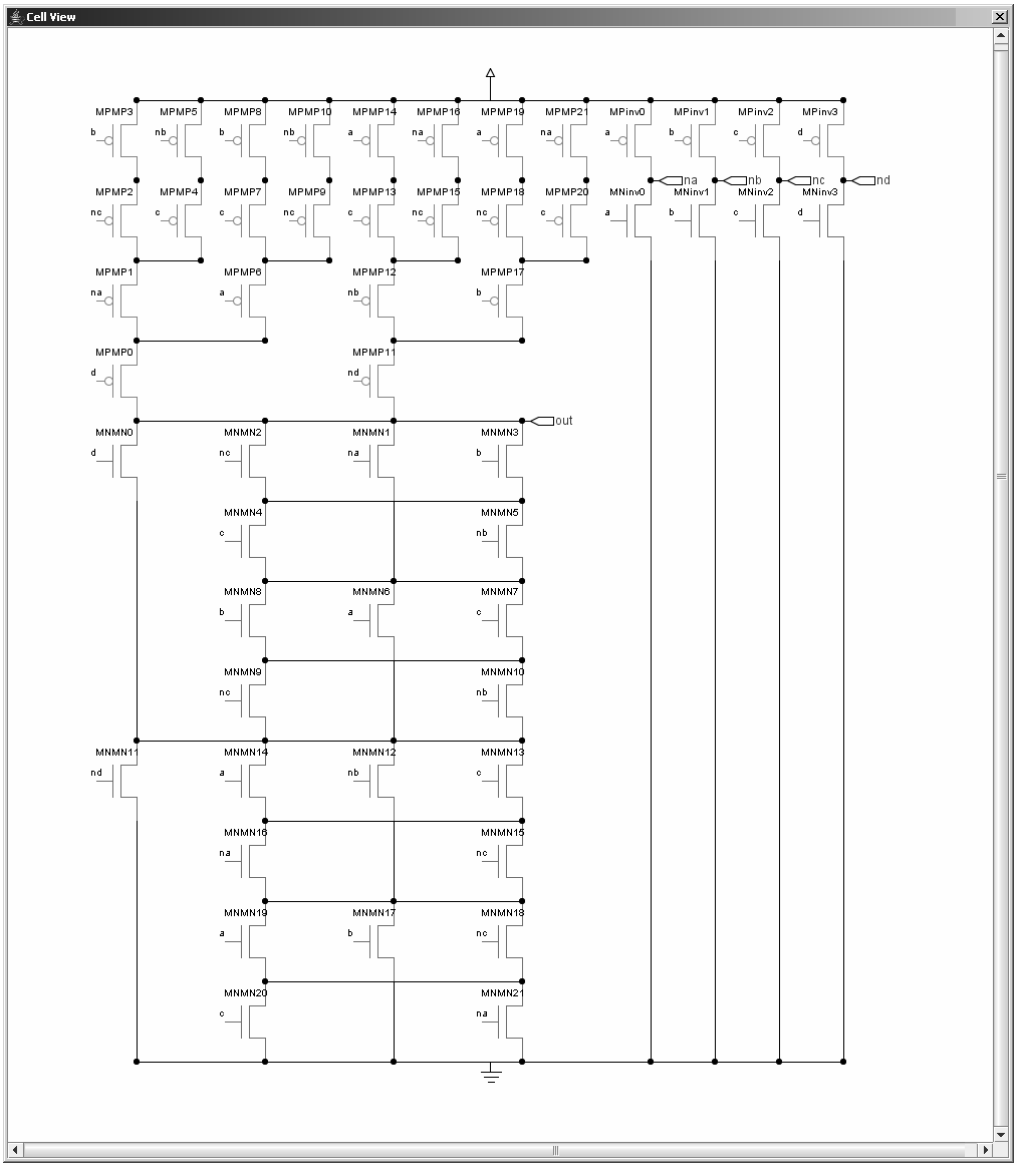


Figure B6: XOR4 in CSP logic style.

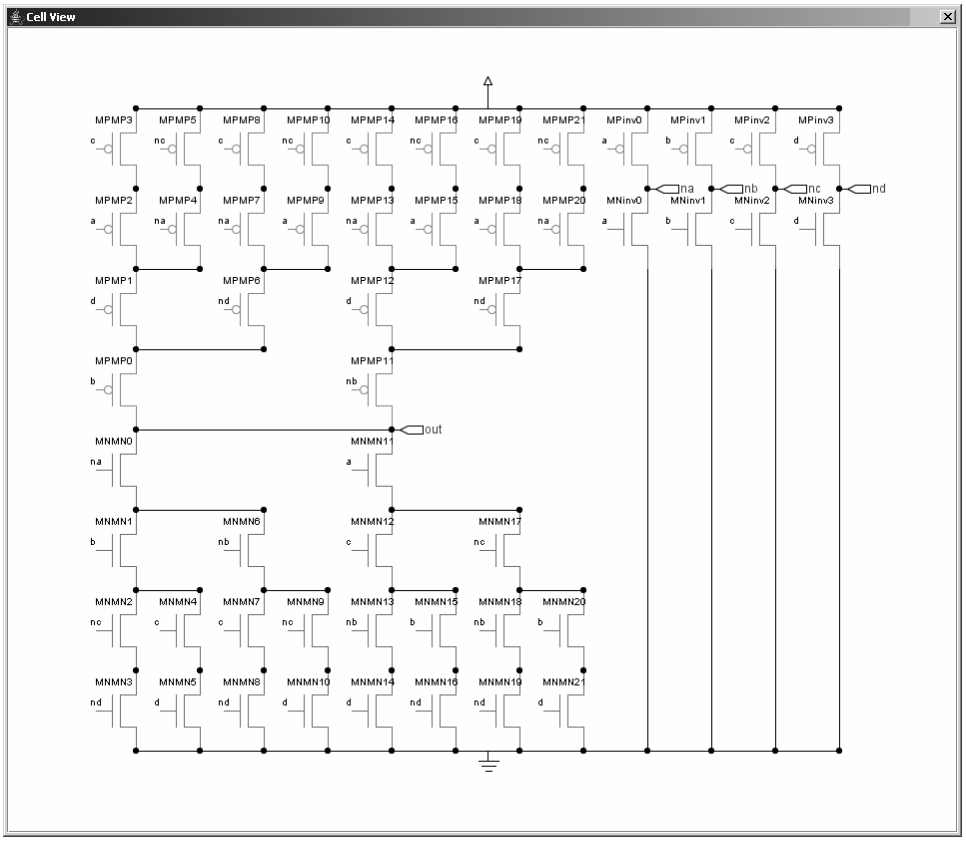


Figure B8: XOR4 in NCSP logic style.

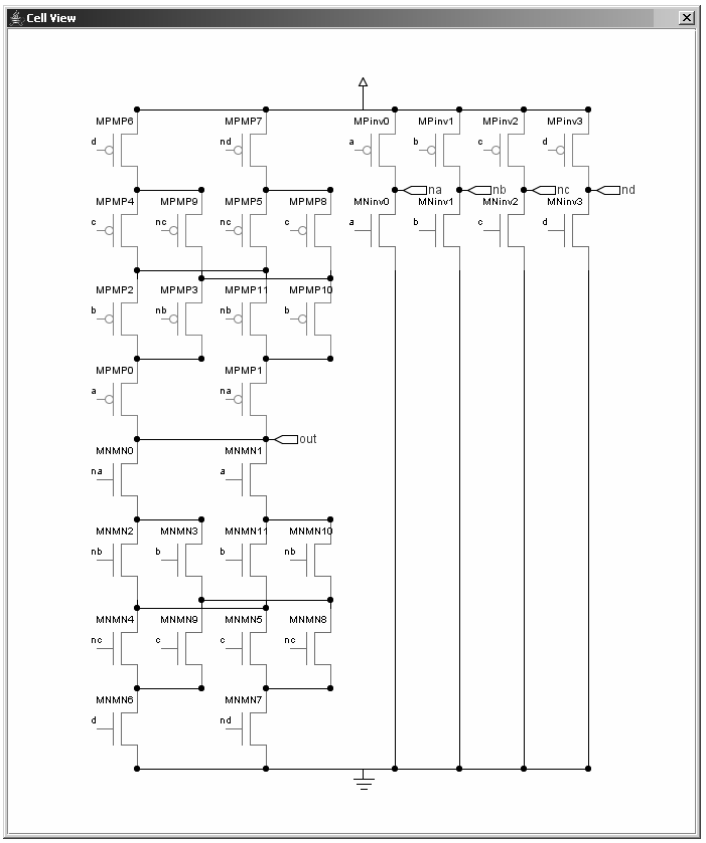


Figure B9: XOR4 in BDD, OpBDD and LBBDD logic styles.

APPENDIX C COUT_FA TRANSISTOR SCHEMATICS

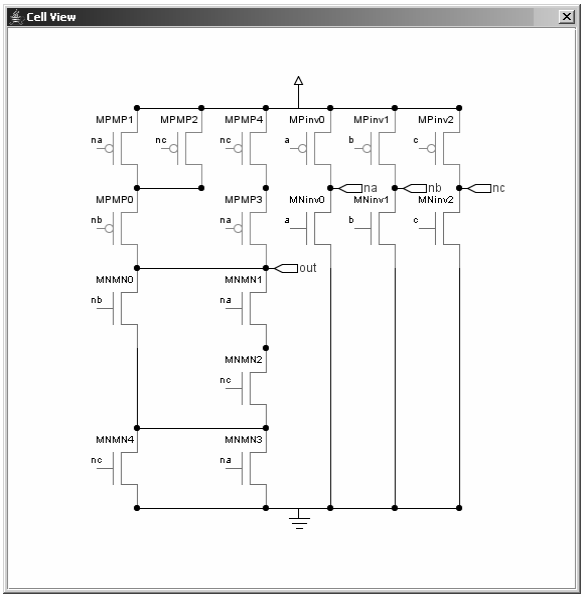


Figure C1: COUT FA in CSP logic style.

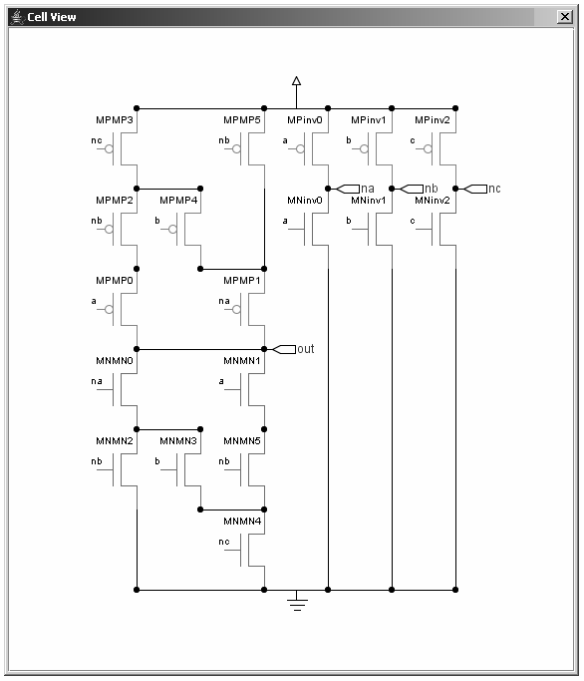


Figure C3: COUT FA in BDD logic style.

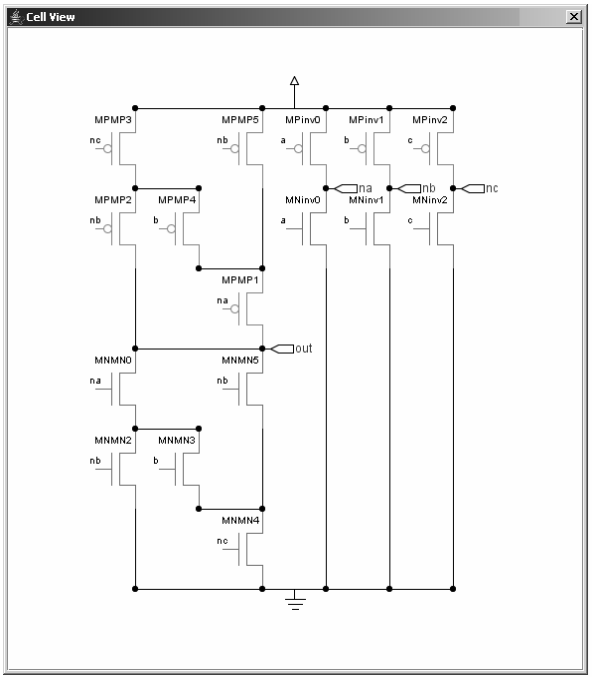


Figure C4: COUT FA in OpBDD logic style.

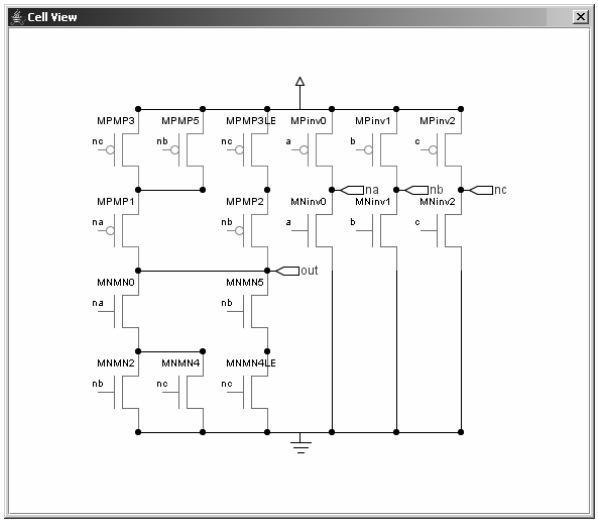


Figure C5: COUT FA in NCSP and LBBDD logic styles.

APPENDIX D LOGIC FUNCTIONS USED FOR THE EXPERIMENTAL RESULTS

Genlib 44-6 up to 4 inputs:

!a
!(a*b)
!(a+b)
!(a*(b+c))
!(a*b*c)
!(a+b*c)
!(a+b+c)
!(a*(b+c*d))
!(a*(b+c+d))
!((a+b)*(c+d))
!(a*b*(c+d))
!(a*b*c*d)
!(a+b*(c+d))
!(a+b*c*d)
!(a*b+c*d)
!(a+b+c*d)
!(a+b+c+d)

Additional logic functions:

(!a*b)+(a*!b)
(!a*(!b*c+b*!c))+(a*(b*c+!b*!c))
(((!c+!b)*!d*(c+b)+(c+!b)*d*(b+!c))*a+((!c+!b)*d*(c+b)+(c+!b)*!d*(b+!c))*!a)!(a+b*(c+d))
(a*b)+(a*c)+(b*c)

500 NPN-classes logic functions up to 5 inputs:

000006B3	000019EE	0001019F	00010AE2	00011BEF
000006B4	000019EF	000101A6	00010AE3	00011BF0
000006B5	000019F1	000101A9	00010AE4	00011BF1
000007BF	000019F3	000101AA	00010AE5	00011BF2
000007E0	000019F6	000101AB	00010AE6	00011BF3
000007E1	000019F7	000101AC	00010AE7	00011BF4
000007E2	000019F8	000101AD	00010AE8	00011BF5
000007E3	000019F9	000101AE	00010AE9	00011BF6
000007E6	000019FA	000101AF	00010AEA	00011BF7
000007E7	000019FB	000101BC	00010AEB	00011BF8

000007E9	000019FE	000101BD	00010AEC	00011BF9
000007EB	000019FF	000101BE	00010AED	00011BFA
000007EF	00001BD6	000101BF	00010AEE	00011BFB
000007F0	00001BD7	000101E8	00010AEF	00011BFC
000007F1	00001BD8	000101E9	00010AF0	00011BFD
000007F2	00001BD9	000109E1	00010AF1	00011BFE
000007F3	00001BDB	000109E2	00010AF2	00011BFF
000007F6	00001BDE	000109E3	00010AF3	00011EE0
000007F7	00001BDF	000109E6	00010AF4	00011EE1
000007F8	00001BE4	000109E7	00010AF5	00011EE2
000007F9	00001BE5	000109E8	00010AF6	00011EE3
000007FA	00001BE7	000109E9	00010AF7	00011EE6
000007FB	00001BEC	000109EA	00010AF8	00011EE7
000007FE	00001BED	000109EB	00010AF9	00011EE8
000007FF	00001BEE	000109EE	00010AFA	00011EE9
00000FF0	00001BEF	000109EF	00010AFB	00011EEA
00000FF1	00001BFC	000109F0	00010AFC	00011EEB
00000FF3	00001BFD	000109F1	00010AFD	00011EEE
00000FF6	00001BFF	000109F2	00010AFE	00011EEF
00000FF7	00001EE1	000109F3	00010AFF	00011EF0
00000FFF	00001EE3	000109F6	00010BB0	00011EF1
00001668	00001EE6	000109F7	00010BB1	00011EF2
00001669	00001EE7	000109F8	00010BB2	00011EF3
0000166A	00001EE9	000109F9	00010BB3	00011EF6
0000166B	00001EEB	000109FA	00010BB4	00011EF7
0000166E	00001EEE	000109FB	00010BB5	00011EF8
0000166F	00001EEF	000109FE	00010BB6	00011EF9
0000167E	00001EF1	000109FF	00010BB7	00011EFA
0000167F	00001EF3	00010AA0	00010BB8	00011EFB
00001681	00001EF6	00010AA1	00010BB9	00011EFE
00001683	00001EF7	00010AA2	00010BBA	00011EFF
00001686	00001EF9	00010AA3	00010BBB	00011FF0
00001687	00001EFA	00010AA4	00010BBC	00011FF1
00001689	00001EFB	00010AA5	00010BBD	00011FF2
0000168B	00001EFE	00010AA6	00010BBE	00011FF3
0000168E	00001EFF	00010AA7	00010BBF	00011FF6
0000168F	00001FF1	00010AA8	00010BD0	00011FF7
00001696	00001FF2	00010AA9	00010BD1	00011FF8
00001697	00001FF3	00010AAA	00010BD2	00011FF9
00001698	00001FF6	00010AAB	00010BD3	00011FFA
00001699	00001FF7	00010AAC	00010BD6	00011FFB
0000169A	00001FF8	00010AAD	00010BD7	00011FFE
0000169B	00001FF9	00010AAE	00010BD8	00011FFF
0000169E	00001FFA	00010AAF	00010BD9	00012880
0000169F	00001FFB	00010AB0	00011AFD	00012881
000016A9	00001FFE	00010AB1	00011AFE	00012882
000016AB	00001FFF	00010AB2	00011AFF	00012883
000016AC	00003CC3	00010AB3	00011BB0	00012884
000016AD	00003CC7	00010AB4	00011BB1	00012885
000016AE	00003CCF	00010AB5	00011BB2	00012886
000016AF	00003CD7	00010AB6	00011BB3	00012887
000016BC	00003CDB	00010AB7	00011BB4	00012888

000016BD	00003CDF	00010AB8	00011BB5	00012889
000016BE	00003CFF	00010AB9	00011BB6	0001288A
00001796	00003DD6	00010ABA	00011BB7	0001288B
00001797	00003DD7	00010ABB	00011BB8	0001288C
00001798	00003DDA	00010ABC	00011BB9	0001288D
00001799	00003DDB	00010ABD	00011BBA	0001288E
0000179A	00003DDE	00010ABE	00011BBB	0001288F
0000179B	00003DDF	00010ABF	00011BBC	00012894
0000179E	00003DED	00010AC0	00011BBD	00012895
0000179F	00003DEF	00010AC1	00011BBE	00012896
000017A9	00003DFD	00010AC2	00011BBF	00012897
000017AB	00003DFE	00010AC3	00011BD0	00012898
000017AC	00003DFF	00010AC6	00011BD1	00012899
000017AD	00003FFC	00010AC7	00011BD2	0001289A
000017AE	00003FFD	00010AC8	00011BD3	000128AA
000017AF	00003FFF	00010AC9	00011BD6	000128AB
000017BC	00006996	00010ACA	00011BD7	000128AC
000017BD	0001017E	00010ACB	00011BD8	000128AD
000017BE	0001017F	00010ACE	00011BD9	000128AE
000017BF	00010180	00010ACF	00011BDA	000128AF
000017E8	00010181	00010AD0	00011BDB	000128BC
000017E9	00010182	00010AD1	00011BDE	000128BD
000017EA	00010183	00010AD2	00011BDF	000128BE
000017EB	00010186	00010AD3	00011BE0	000128BF
000017EE	00010187	00010AD4	00011BE1	000128C0
000017EF	00010188	00010AD5	00011BE2	000128C1
000017FE	00010189	00010AD6	00011BE3	000128C2
000017FF	0001018A	00010AD7	00011BE4	000128C3
000018E7	0001018B	00010AD8	00011BE5	000128C4
000018EF	0001018E	00010AD9	00011BE6	000128C5
000018FF	0001018F	00010ADA	00011BE7	000128C6
000019E1	00010196	00010ADB	00011BE8	000128C7
000019E3	00010197	00010ADC	00011BE9	000128CA
000019E6	00010198	00010ADD	00011BEA	000128CB
000019E7	00010199	00010ADE	00011BEB	012CDF18
000019E9	0001019A	00010ADF	00011BEC	012CDF19
000019EA	0001019B	00010AE0	00011BED	012CDF1A
000019EB	0001019E	00010AE1	00011BEE	012CDF2A

7 Logic functions unfeasible in CSP:

000101170117173F
000101170117177F
011313370337377F
011313371337377F
011313371337777F
0117177F177F7FFF
0117177F577F7FFF

Branch-based vs. factorized logic functions:

!a	!a
!(a*b)	!(a*b)
!(a+b)	!(a+b)
!(a*(b+c))	!((a*b)+(a*c))
!(a*b*c)	!(a*b*c)
!(a+b*c)	!(a+b*c)
!(a+b+c)	!(a+b+c)
!(a*(b+c*d))	!((a*c*d)+(a*b))
!(a*(b+c+d))	!((a*d)+(a*c)+(a*b))
!((a+b)*(c+d))	!((b*d)+(b*c)+(a*d)+(a*c))
!(a*b*(c+d))	!((a*b*d)+(a*b*c))
!(a*b*c*d)	!(a*b*c*d)
!(a+b*(c+d))	!((b*d)+(b*c)+(a))
!(a+b*c*d)	!(a+b*c*d)
!(a*b+c*d)	!(a*b+c*d)
!(a+b+c*d)	!(a+b+c*d)

APPENDIX E DEVELOPED TOOLS

During the development of this work it became evident the need for a logic synthesis tool that could be used to optimize logic descriptions, to generate and evaluate transistor networks, and to estimate some electrical and physical behaviors at logic cells level. In fact, there are not many available tools in the academy that could be used to perform some of these tasks. An example is the SIS tool from Berkley (SENTOVICH, 1992), which is a technology mapping tool, that presents scripts to perform factorization, for instance. More recently, its new brother, called ABC, has incorporated some old features and presents new algorithms and techniques for technology mapping purpose (MISCHENKO, 2005). Other examples of CAD tools are: CDF, which provides the generation of transistor cells layout from an input equation description, performing some intermediate logic synthesis steps (NANGATE, 2008); Blue Macaw Didactic Placement Tool, which is an environment to experiment and to learn the existing VLSI Cell Placement Algorithms and their variations (HENTSCHKE, 2008); Cellgen that is a tool for generating cell layout from a predefined input spice netlist (ZIESEMER JUNIOR, 2007). However, these available tools do not provide a wide cover for logic synthesis problems, or, at least, they do not permit the use of a single and specific operation. On the other hand, several small tools and scripts are available in Nangate-UFRGS Research Lab. The majority of these codes and modules were developed for internal purposes, to support and to assist the development of researches. A repository and a version control system allow students and researchers to share codes and to optimize the process of generating new solutions and results.

In this context, an academic environment composed of 3 parts was idealized by our group. The first one is a logic synthesis tool, which only performs logic manipulations over Boolean functions. This tool is called KARMA 3. The second one is an electrical synthesis tool, developed to provide transistor networks generation, manipulation and evaluation at cell level. This tool is named ELECTRO. The last one is a physical synthesis tool, proposed to implement, optimize and evaluate the layout of logic cells. This tool, named LAGARTO, is under development and will complete the logic cell automated design flow. The produced code herein is presented in part of the KARMA 3, and it is the main engine for the ELECTRO tool. Both tools were developed in Java language and they will be presented in the following sub sections.

A Tool for Logic Synthesis of Boolean Functions

KARMA 3 tool is a new and expanded version from the original KARMA tool (KARMA, 2008). The main objective of KARMA was to help users to have a better understanding of Karnaugh maps, truth-tables, Boolean functions and many concepts of logic synthesis. Through a friendly graphical user interface the user can interact with the software in many ways like assigning a Boolean function and running Quine-McCluskey algorithm to get the prime implicants and see how they are positioned in the Karnaugh map.

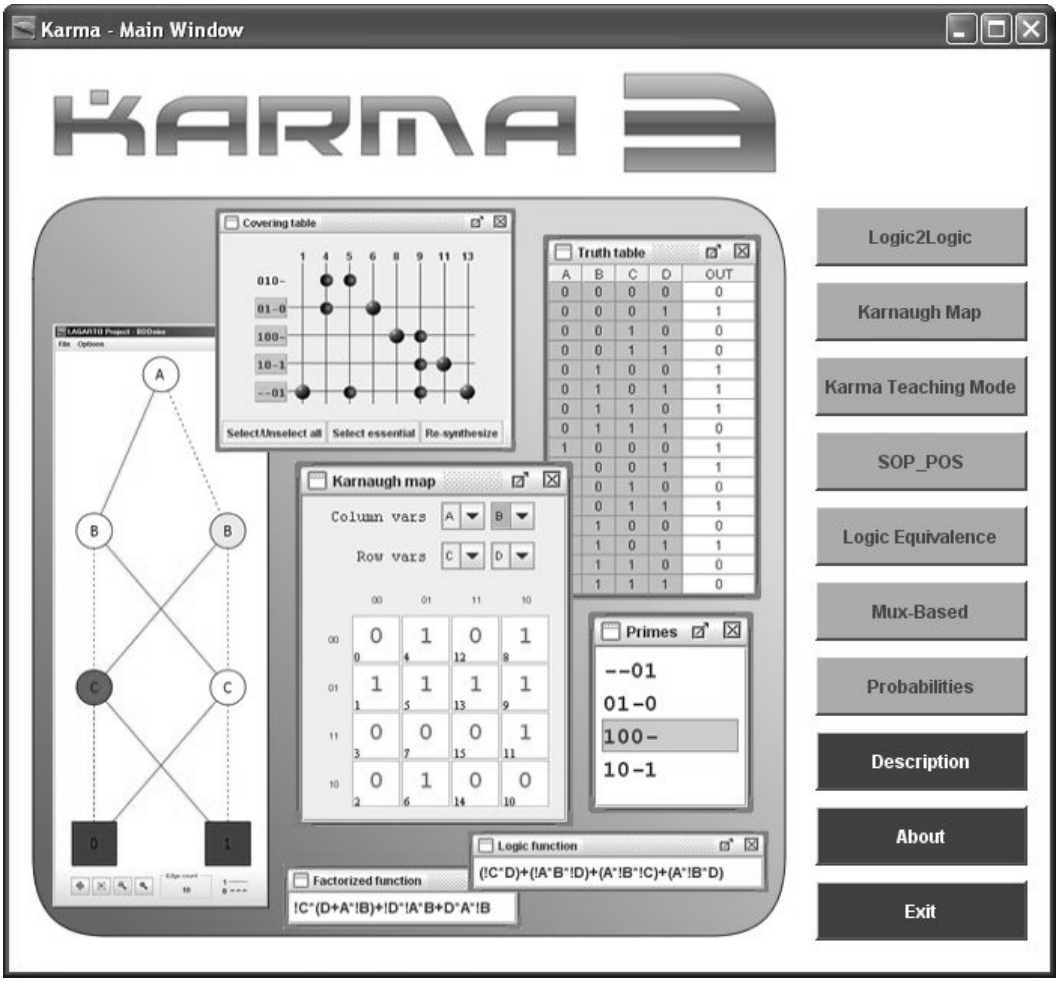


Figure E1: KARMA 3 main window.

The original version was rebuilt to enlarge the number of features and data manipulations in the logic synthesis field. The current version contains a converter for different logic descriptions, a logic equivalence verifier, an SOP and POS generator and analyzer, a factorization unit, and a probability evaluator of signal propagation. In other words, KARMA 3 is a tool that offers the possibility of manipulating Boolean functions

to meet the designer needs in terms of logic description. Figure E1 presents the main window of KARMA 3.

The first module is the "*Logic2Logic*". The purpose of this module is to translate a given logic description to another. It is possible to convert BLIF description, truth-table representation, equation description, numerical description or list of minterms representation in between them. Figure E2 shows the window of this module.

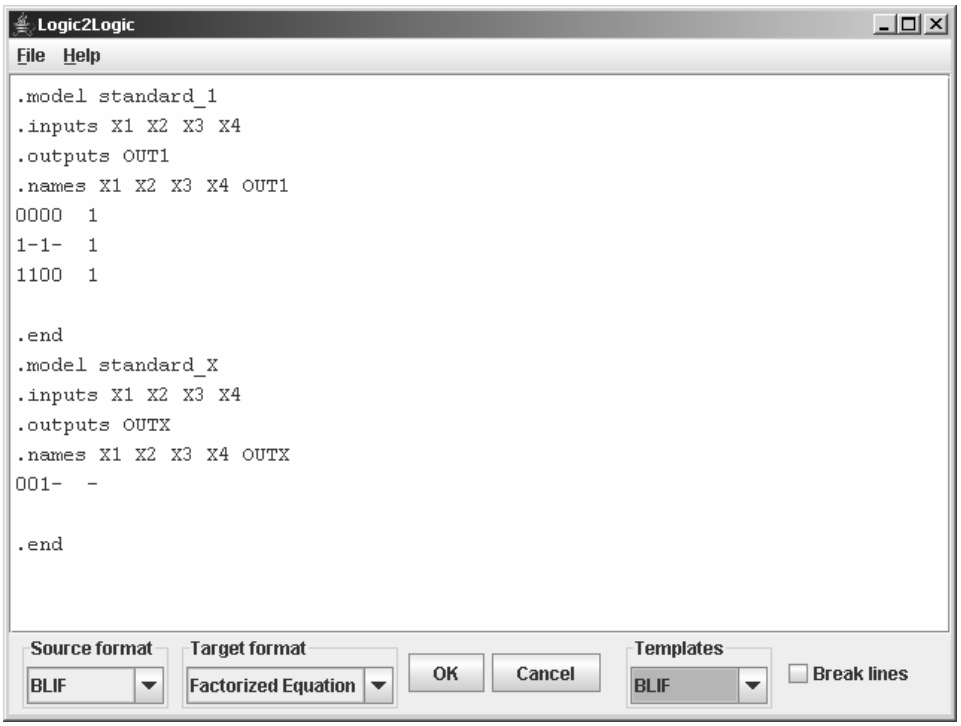


Figure E2: Logic2Logic window.

The "*Karnaugh Map*" module permits to synthesize using Karnaugh maps. It is possible to set the function through truth-table or directly in the map, and then run the Quine-McCluskey algorithm. From this point, the list of prime implicants, the Quine-McCluskey step-by-step procedure, the covering table, and the equivalent factorized function can be visualized. Figure E3 illustrates the window of this module.

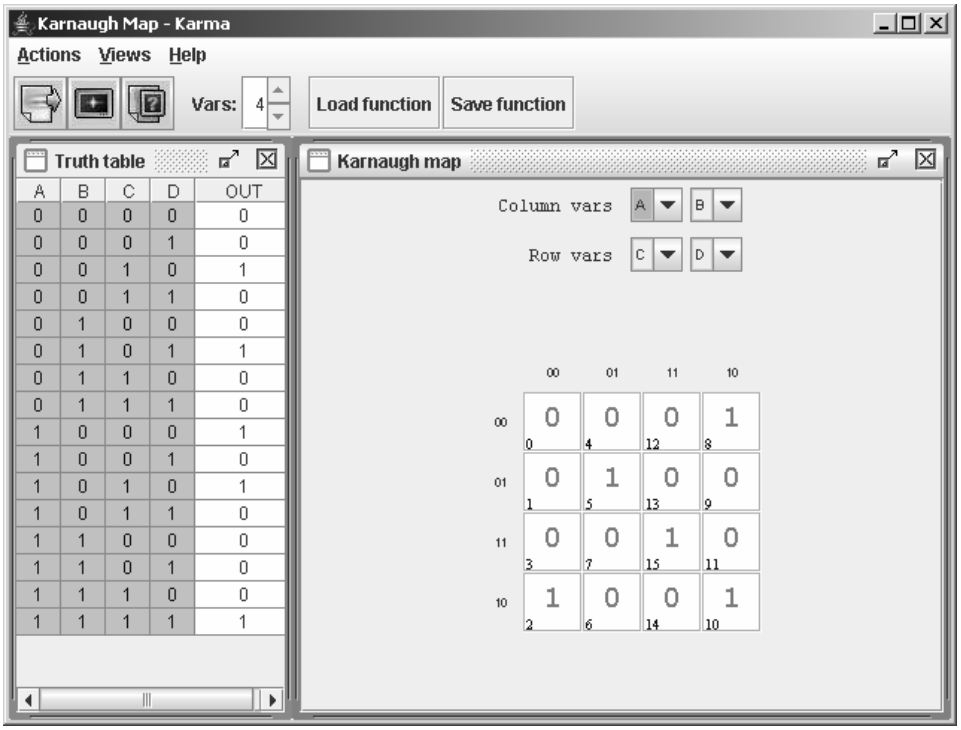


Figure E3: Karnaugh Map window.

In the "*Karma Teaching Mode*" some games are presented to be used as a didactic instrument. In this module the user learn and perform some exercises about Karnaugh maps, viewing and finding adjacent minterms and cubes, selecting cubes, and covering table. Figure E4 illustrates this module.

The "*SOP_POS*" module delivers an easy way to obtain the sum-of-products and the product-of-sums from a given logic input description. Figure E5 shows the window of this module.

In the "*Logic Equivalence*" module, illustrated in Figure E6, it is possible to evaluate if two different logic descriptions are equivalent or not. This equivalence is done through BDD evaluation, and it returns the truth-table for the input functions described by the user.

The "*Mux-Based*" module evaluates all possible configurations of using variables as pass variables or control variables, and informs the best solution to implement a transistor cell with a reduced number of elements in a mux-based logic gate, as presented in section 3.2.1.3. Figure E7 shows this feature.

The last module, called "*Probabilities*", permits the user to determine the probability of occurring a one logic value in the output of a given logic function according to the occurrence probabilities in the inputs. This module is depicted in Figure E8.

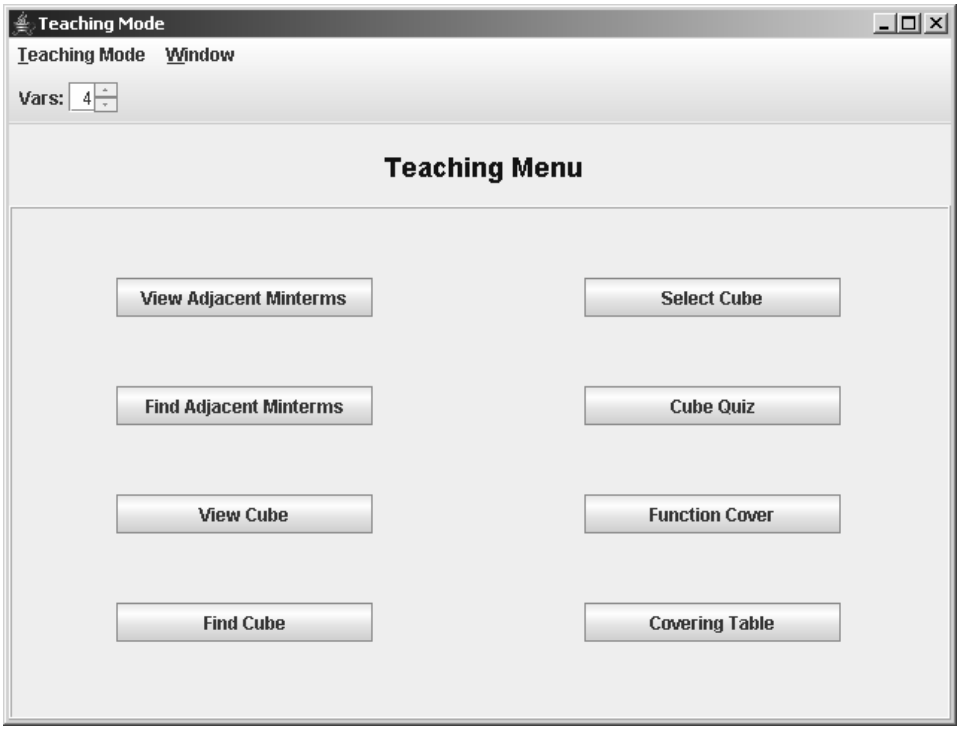


Figure E4: Karma teaching mode window.

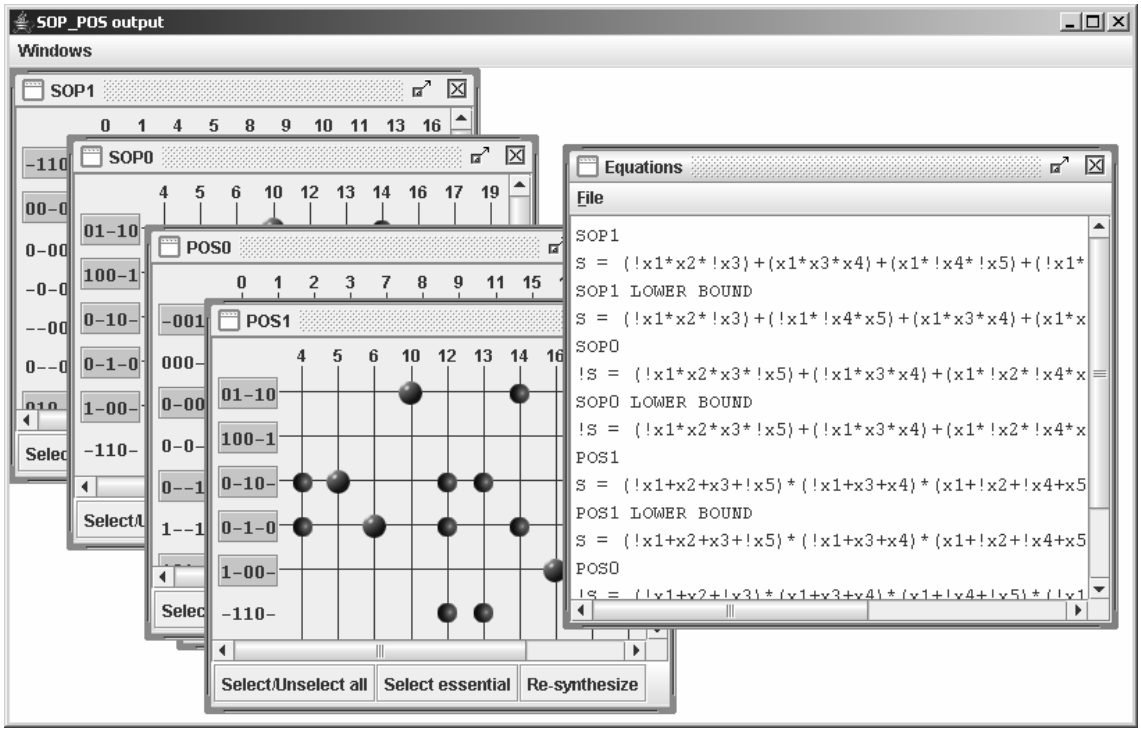


Figure E5: SOP_POS window.

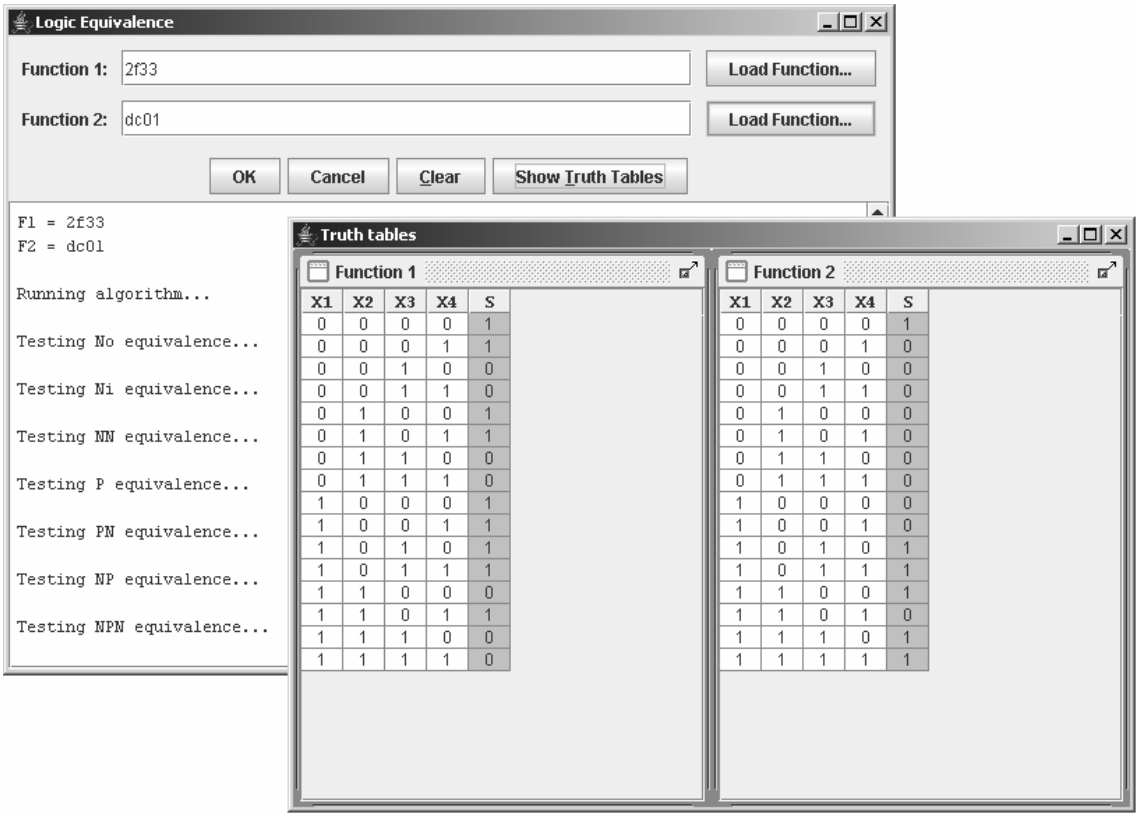


Figure E6: Logic equivalence window.

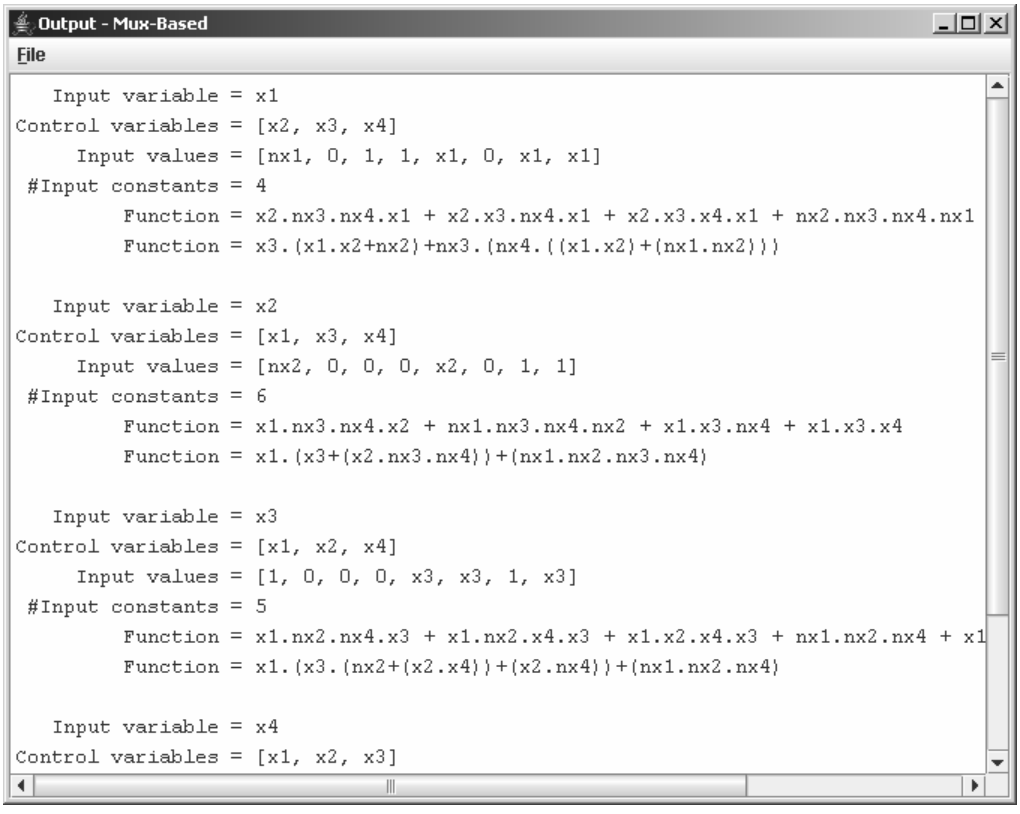


Figure E7: Mux-based window.

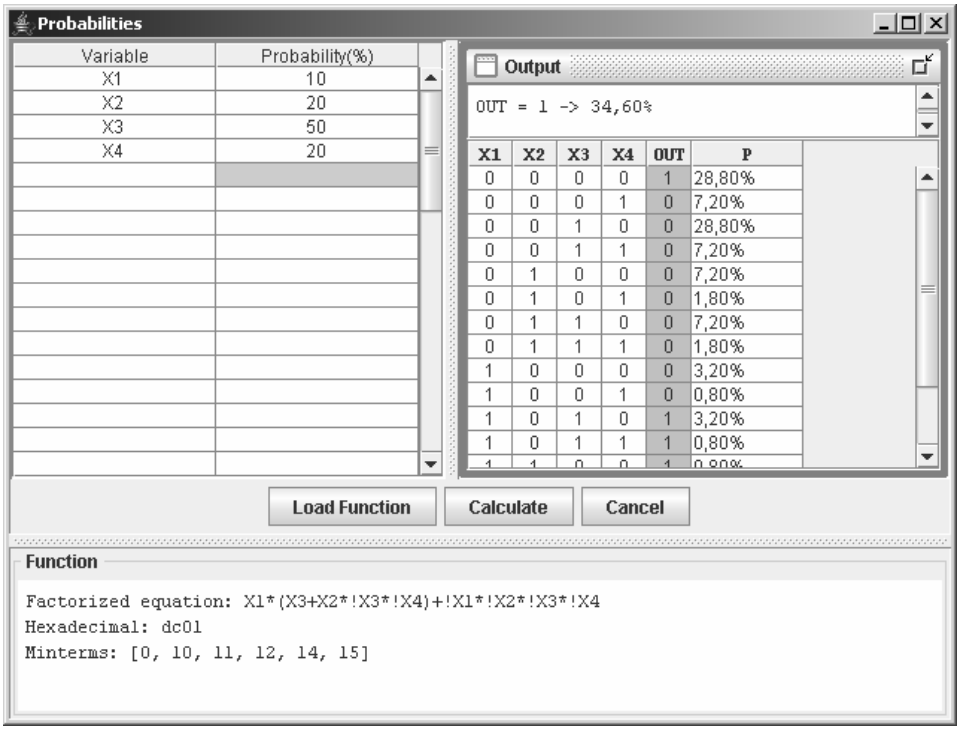


Figure E8: Probabilities window.

All modules of KARMA 3 were developed together with other students and researchers in the Nangate-UFRGS Research Lab.

A Tool for Automatic Transistor Cells Generation

ELECTRO tool presents some features that permit the user to generate transistor networks and to evaluate some electrical and physical characteristics through estimations. It is a fast and easy way to investigate the behavior of possible transistor networks implementations for a given input logic function. As input, this tool accepts a Boolean expression description. Also, it is possible to use a Spice netlist input, allowing the user to manipulate and to evaluate a pre-implemented transistor network.

This tool contains several algorithms and methods implemented during the development of this work. The following list describes some of them, since the tool is constantly being improved:

- BDD ordering methods
  - Sifiting algorithm
  - Exhaustive solution
- Transistor network generation
  - CSP logic
  - NCSP logic

    - BDD logic (CMOS derived from BDD)
    - OpBDD logic (Optimized BDD logic)
    - LBBDD (Lower Bound BDD logic)
    - Mux-based logic
    - Branch-based logic
    - Dual-graph logic
- Transistor network simulator
- Network profile algorithms
    - Number of nodes
    - Number of transistors
    - Number of transistors connected per node
    - Number of unate/binate nodes
    - Number of inverters
    - Number of paths in the transistor networks
    - Number of paths in the BDD
    - Number of transistor in series in the network
    - Smallest and largest paths in the transistor network
- Leakage estimation methods
    - Gate leakage
    - Subthreshold leakage
    - Iterative gate/subthreshold leakage
- Transistor sizing
    - Logical Effort method
    - Fixed size (minimal or relative to some sizing rule)
- Transistor folding
- Network transistor sharing
- Structural transistor ordering
- Variables occurrence in transistor networks
- Detection of series-parallel or bridge transistor arrangements

The main window from ELECTRO, shown in Figure E9, follows a similar graphical user interface presented in KARMA 3. All options are grouped in a list according to their functionalities.

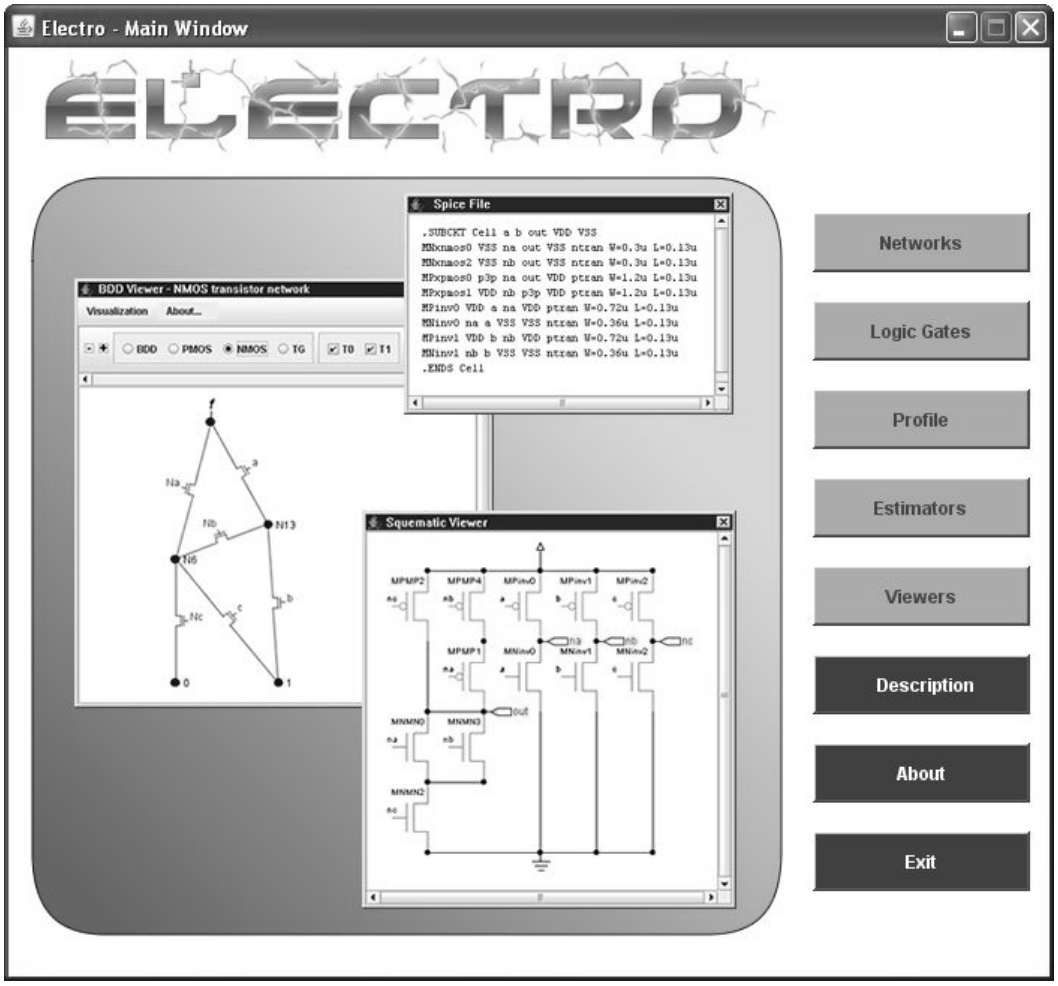


Figure E9: ELECTRO main window.

The "*Networks*" module is a small tool that permits to generate different switch networks. It is a powerful didactic instrument for teaching switch theory. In this module it is possible to generate switch networks composed by NMOS transistor, PMOS transistors, mixed NMOS/PMOS transistors and/or transmition gates. The user can generate disjoint pull-up and/or pull-down planes in separated steps, or non-disjoint planes, using or not, mixed NMOS/PMOS transistors. The dual-graph transistor generation concept is presented in this module in order to demonstrate how it is possible to obtain a topologically complementary network from a dual graph.

The "*Logic Gate*" module contains the CMOS logic gates generation. The user can choose one logic style from the large available list. All logic styles presented in this thesis are present in this module. Also, some other logic styles, like PTL, are available. In the future, we will expand it by adding dynamic logic styles.

In the "*Sizing*" module the user can perform transistor sizing in the network. The Logical Effort method is available (SUTHERLAND, 1999). Also, a fixed transistor sizing option is offered, which allows the use of minimal or a relative transistor sizing according to the input rules.

"*Profile*" is the module that delivers all information about the transistor networks that do not need to be simulated or estimated. For a given transistor networks the user can extract information about transistors, nodes, chains and internal transistor arrangements. It is also possible to investigate some BDD structure characteristics, like the BDD sizing and the node unateness property.

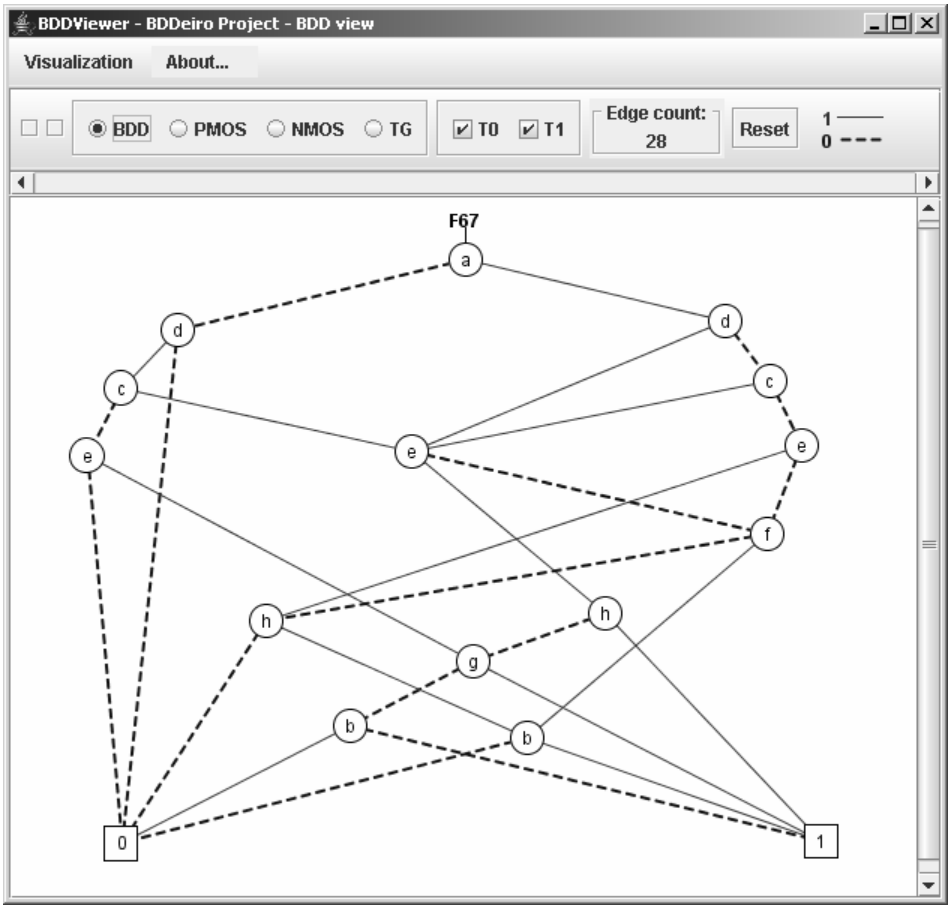


Figure E10: BDD viewer.

The "*Estimation*" module includes all developed estimation methods described in chapter 4. For a given logic gate, the user can evaluate the static power characteristics (the gate leakage behavior, the subthreshold leakage, and the interaction of these two leakage components), the dynamic power characteristic, the delay performance and some physical information. In other words, this module contains information that cannot be extracted by a simple analysis of the network. It includes state-of-the-art methods to evaluate area, delay and power characteristics from a certain logic gate.

Finally, the "*Viewer*" module includes two graphical viewer tools. The first one is dedicated to BDDs. The user can visualize and manipulate the generated BDD that represents the logic function. The second is dedicated to the Spice netlist description. A schematic transistor view is generated to make easy the visualization of transistor

arrangements. Figure E10 and Figure E11 illustrate a BDD representation and an electrical schematic from a given logic function, both generated by the tool.

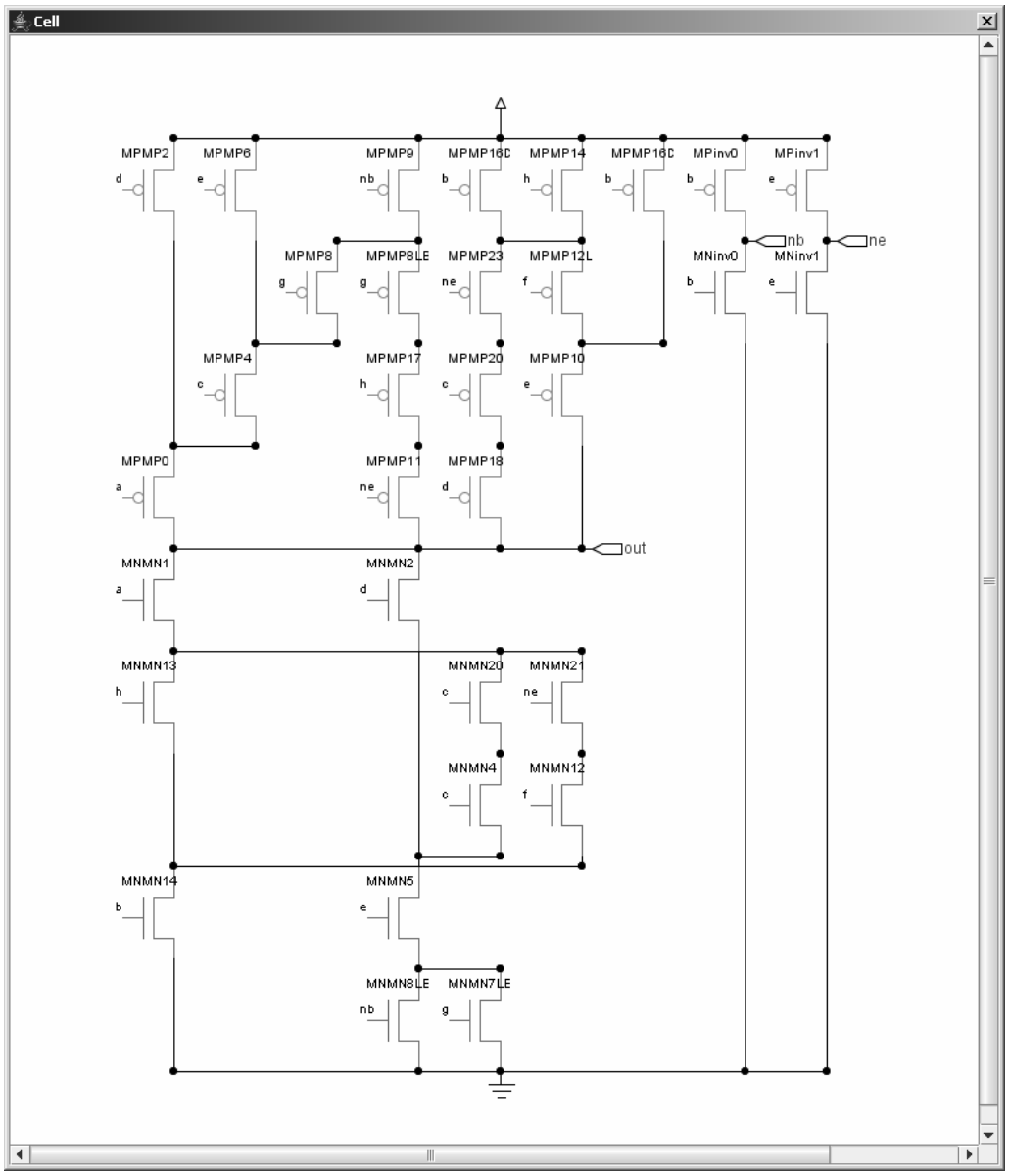


Figure E11: Schematic viewer.

APPENDIX F LIST OF PUBLICATIONS

CARDOSO, T. M. G.; DA ROSA JUNIOR, Leomar Soares; MARQUES, F. S.; RIBAS, R. P.; REIS, André Inácio. Speed-up of ASICs Derived from FPGAs by Transistor Network Synthesis Including Reordering. In: 9th IEEE International Symposium on Quality Electronic Design (ISQED), 2008, San Jose. 9th IEEE International Symposium on Quality Electronic Design Proceedings. Los Alamitos : IEEE Computer Society, 2008. p. 47-52.

BUTZEN, P. F.; DA ROSA JUNIOR, Leomar Soares; CHIAPPETTA FILHO, E. J. D.; MOURA, D.; REIS, André Inácio; RIBAS, R. P. Simple and accurate method for fast static current estimation in CMOS complex gates with interaction of leakage mechanisms. In: 18th ACM Great Lakes Symposium on VLSI (GLSVLSI), 2008, Orlando. Proceedings of the 18th ACM Great Lakes Symposium on VLSI. New York : ACM, 2008. p. 407-410.

BUTZEN, P. F.; DA ROSA JUNIOR, Leomar Soares; CHIAPPETTA FILHO, E. J. D.; MOURA, D.; REIS, André Inácio; RIBAS, R. P. Subthreshold and Gate Leakage Estimation in Complex Gates. In: 17th ACM/IEEE International Workshop on Logic and Synthesis (IWLS), 2008, Lake Tahoe. Workshop Notes of the 17th ACM/IEEE International Workshop on Logic and Synthesis, 2008.

DA ROSA JUNIOR, Leomar Soares; MARQUES, F. S.; SCHNEIDER, F.; RIBAS, R. P.; REIS, André Inácio. A Comparative Study of CMOS Gates with Minimum Transistor Stacks. In: 20th ACM Symposium on Integrated Circuits and Systems Design (SBCCI), 2007, Rio de Janeiro. 20th ACM Symposium on Integrated Circuits and Systems Design Proceedings. New York : ACM, 2007. p. 93-98.

BUTZEN, P. F.; MANCUSO, R.; SCHNEIDER, F.; DA ROSA JUNIOR, Leomar Soares; REIS, André Inácio; RIBAS, R. P. Leakage Behavior in CMOS and PTL Logic Styles for Logic Synthesis Orientation. In: 16th ACM/IEEE International Workshop on Logic and Synthesis (IWLS), 2007, San Diego. Workshop Notes of the 16th ACM/IEEE International Workshop on Logic and Synthesis, 2007. p. 53-58.

GOMES, M. V.; SILVA, C. A.; BAVARESCO, S.; SARTORI, G. H.; DA ROSA JUNIOR, Leomar Soares; REIS, André Inácio; RIBAS, R. P. Test Circuit for Functional Verification of Automatically Generated Cell Library. In: 8th IEEE Latin American Test Workshop (LATW), 2007, Cuzco. Workshop Notes of the 8th IEEE Latin American Test Workshop, 2007.

DA ROSA JUNIOR, Leomar Soares; SCHNEIDER, F.; RIBAS, R. P.; REIS, André Inácio. Analysis of Transistor Networks Generation. In: XIII Workshop Iberchip (IWS),

2007, Lima. Workshop Notes of the XIII Workshop Iberchip. Lima : Editorial Hozlo S.R.L, 2007. p. 383-386.

MARQUES, F. S.; DA ROSA JUNIOR, Leomar Soares; RIBAS, R. P.; SAPATNEKAR, S.; REIS, André Inácio. DAG based library-free technology mapping. In: 17th ACM Great Lakes Symposium on VLSI (GLSVLSI), 2007, Lago Maggiore. 17th ACM Great Lakes Symposium on VLSI Proceedings. New York : ACM, 2007. p. 293-298.

GOMES, M. V.; SILVA, C. A.; BAVARESCO, S.; ALEGRETTI, C.; SARTORI, G. H.; DA ROSA JUNIOR, Leomar Soares; REIS, André Inácio; RIBAS, R. P. Test Circuit for Functional Verification of Automatically Generated Cell Library. In: 12th IEEE European Test Symposium (ETS), 2007, Freiburg. 12th IEEE European Test Symposium Informal Digest of Papers, 2007. p. 101-104.

DA ROSA JUNIOR, Leomar Soares; MARQUES, F. S.; CARDOSO, T. M. G.; RIBAS, R. P.; SAPATNEKAR, S.; REIS, André Inácio. Fast Disjoint Transistor Networks from BDDs. In: 19th ACM Symposium on Integrated Circuits and Systems Design (SBCCI), 2006, Ouro Preto. 19th ACM Symposium on Integrated Circuits and Systems Design Proceedings. New York : ACM, 2006. p. 137-142.

DA ROSA JUNIOR, Leomar Soares; MARQUES, F. S.; CARDOSO, T. M. G.; RIBAS, R. P.; REIS, André Inácio. BDDs and transistor networks with minimum pull-up/pull-down chains. In: 15th ACM/IEEE International Workshop on Logic and Synthesis (IWLS), 2006, Vail. Workshop Notes of the 15th ACM/IEEE International Workshop on Logic and Synthesis, 2006. p. 142-149.

APPENDIX G GERAÇÃO AUTOMÁTICA E AVALIAÇÃO DE REDES DE TRANSISTORES EM DIFERENTES ESTILOS LÓGICOS

Os circuitos digitais estão cada vez mais presentes no dia-a-dia da vida moderna causando um amplo impacto na sociedade. Esse impacto se deve ao fato de que circuitos digitais se aplicam diretamente ou auxiliam diferentes áreas do conhecimento. Exemplos disso são os computadores pessoais, a telefonia móvel celular, os dispositivos GPS (*Global Positioning System*), os sistemas automotivos computadorizados, a computação em equipamentos e dispositivos da medicina e etc. Esta explosão na presença de circuitos digitais em vários campos do conhecimento pode ser atribuída, em grande parte, ao avanço das tecnologias de concepção de circuitos integrados. Este avanço permite a integração de um número cada vez maior de componentes, possibilitando a concepção de circuitos cada vez maiores e mais complexos. A alta integração e as novas tecnologias de fabricação disponíveis impõem novos limites e desafios para a síntese. As principais dificuldades são a adaptação aos novos parâmetros de tecnologia e o desenvolvimento de projetos em um tempo curto o suficiente para não comprometer a sua comercialização (*time-to-market*). Portanto, a automatização desse processo através do uso intenso de ferramentas de CAD (*Computer Aided Design*) é um fator cada vez mais indispensável para alcançar essas metas. Ao utilizar uma ferramenta de síntese automática, o efeito esperado é a obtenção de resultados de igual ou melhor qualidade que os realizados manualmente, mas em um tempo muito mais curto. Em geral, projetos feitos manualmente são muito mais custosos, mas mais eficientes em termos de área, potência consumida e desempenho. Portanto, prover ferramentas automatizadas para a concepção de circuitos eficientes é um desafio e uma oportunidade que se estabelece para atender a crescente demanda do mercado moderno.

Basicamente existem duas formas de se obter um circuito integrado (DEMICHELLI, 1994). Estas formas são chamadas de fluxos de síntese e estão divididas em *custom* e *semicustom*. Os fluxos de projeto chamados *custom* são aqueles onde todos os passos para obtenção do circuito integrado são executados manualmente pelos projetistas. Este estilo de projeto possibilita uma alta flexibilidade para a obtenção do circuito, uma vez que todas as etapas para geração do circuito final podem ser

exploradas e otimizadas visando à implementação de um circuito de alta qualidade. Contudo, os fluxos de projetos *custom* apresentam um alto custo monetário. Isso se deve, em parte, a necessidade de um número considerável de recursos humanos para atender todas as etapas de desenvolvimento. Projetos *custom* eram muito comuns nos primeiros anos da microeletrônica, quando os projetos não eram muito grandes e não existiam ferramentas de síntese automática disponíveis. Os fluxos de projeto chamados *semicustom* podem ser divididos em dois estilos distintos: baseados em matrizes e baseados em células. O estilo baseado em matrizes utiliza matrizes de elementos configuráveis para descrever a lógica a ser implementada. Um exemplo desse estilo são os FPGAs (*Field Programmable Gate Array*), os quais possuem um conjunto de elementos lógicos interconectados programáveis. O estilo baseado em células é desenvolvido através da utilização de macro células, células padrão ou células geradas automaticamente. As macro células são, em geral, blocos construídos pela união de unidades lógicas menores com alto grau de repetição, como blocos de memória e circuitos aritméticos. Células padrão são pequenas porções de circuito projetadas manualmente ou por ferramentas industriais e acadêmicas (NANGATE, 2008). Essas ferramentas podem servir como geradores ou servidores de células. Quando projetadas manualmente, estas células costumam ser muito eficientes pelos mesmos motivos que os circuitos *custom*. Além disso, são bem caracterizadas, ou seja, suas informações elétricas e de área são conhecidas com boa precisão. As células geradas automaticamente são blocos similares às células padrão, mas com a essencial diferença de terem sido geradas automaticamente por um gerador de células. Segundo Vujkovic (2002), a geração automática de células pode levar a implementação de circuitos com desempenho desejável, se comparado com projetos *custom*. Como células geradas são criadas de forma automática, o conjunto de células disponíveis para a realização do mapeamento do circuito não precisa ser tão restrito como os conjuntos pré-projetados de células padrão. Essa característica, portanto, possibilita ao desenvolvedor um maior grau de liberdade para explorar a realização da etapa de mapeamento tecnológico e a obtenção final do circuito. Contudo, como desvantagem, pode-se citar a complexidade que será agregada na etapa de mapeamento devido ao grande número de células que podem ser disponibilizadas pelo gerador.

O fluxo de geração de circuitos baseados em células, independentemente das células que compõem a biblioteca de células terem sido geradas manualmente ou automaticamente, apresenta como ponto inicial uma descrição comportamental, e como saída o leiaute do circuito final. Segundo Weste (2005), essa transformação se dá em três diferentes domínios. O primeiro deles é o domínio da síntese arquitetural, onde a descrição está no seu mais alto nível e representa uma visão da organização do sistema. As etapas seguintes ocorrem no domínio da síntese lógica, onde as descrições são tratadas com modelos lógicos dos componentes e blocos funcionais. O último domínio é o da síntese física, em que as descrições já estão sob um ponto de vista geométrico próximo do leiaute final do circuito.

A síntese lógica é dividida em transformações independentes e dependentes de tecnologia. As transformações independentes de tecnologia são otimizações nas redes

booleanas que descrevem cada bloco do circuito. Em seguida, a etapa de mapeamento tecnológico transforma toda a rede em uma descrição dependente da tecnologia, traduzindo todo o circuito para um conjunto interconectado de células lógicas implementadas em uma dada tecnologia alvo. A partir disso, as transformações são ditas dependentes de tecnologia e são as últimas no domínio da síntese lógica. São exemplos de otimizações dessa fase o redimensionamento de portas e a duplicação de partes da lógica. Por fim, a síntese física usa essa descrição dependente de tecnologia como entrada para etapas como posicionamento das células e roteamento de sinais. O produto final é um leiaute do circuito que segue a especificação comportamental inicial, pronto para ser fabricado.

O conjunto das células lógicas disponíveis para a etapa de mapeamento tecnológico é chamado de biblioteca de células. Ela representa todas as possibilidades de elementos funcionais daquela tecnologia que podem ser usados para implementar o comportamento especificado pela rede booleana. Para isso, o conjunto de células que define uma biblioteca deve ser capaz de implementar qualquer função necessária. Um exemplo de biblioteca é o conjunto unitário contendo apenas a célula *NAND*, pois qualquer função lógica combinacional pode ser implementada com instâncias dessa porta.

Em um fluxo que utiliza células geradas automaticamente, não se possui um conjunto especificado de células pré-projetadas. Neste caso o gerador de células atua como um servidor de células fornecendo células requisitadas pelo mapeamento tecnológico. Para limitar as células que podem ser utilizadas pelo mapeador são definidas, em geral, restrições topológicas que definem células aceitáveis em uma determinada tecnologia através da limitação de características que impeçam a célula de ter um desempenho aceitável. Tipicamente, o número máximo de transistores em série é limitado ou restrito a um valor aceitável (BHATTACHARYA, 2002). As restrições aplicadas especificam um conjunto de células, e o gerador de células disponível deve ser capaz de gerar qualquer uma dessas. O conjunto de células a serem usadas e que o gerador é capaz de gerar é chamado de biblioteca virtual. Como desvantagem de bibliotecas virtuais pode-se citar o fato delas serem fracamente caracterizadas, pois elas não apresentam informações detalhadas sobre o comportamento elétrico de cada uma de suas células como nas pré-caracterizadas. No entanto, um gerador automático pode ser parametrizável, sendo capaz de produzir várias versões da mesma célula com, por exemplo, diferentes tamanhos de transistor. Esta característica pode ter influência no projeto de um circuito com restrições de *timming* (VUJKOVIC, 2002). Outra vantagem é a rapidez de adaptação a uma nova tecnologia. O tempo de reconfiguração de um gerador de células para as novas regras de fabricação é consideravelmente menor que a reconstrução de uma biblioteca pré-caracterizada na nova tecnologia. Essa característica é uma das principais motivações dos projetos orientados a geradores automáticos de células.

Um atributo interessante para a utilização de um gerador automático de células é o fato de que diversos estilos lógicos podem ser implementados automaticamente. Em

geral, bibliotecas de células pré-caracterizadas são compostas por células implementadas em um único estilo lógico, como, por exemplo, CMOS (*Complementary Metal Oxide Semiconductor*). Utilizando-se um gerador automático capaz de prover células lógicas em diversos estilos lógicos, dá-se a liberdade para que projetistas de circuitos integrados explorem, ainda mais, o espaço de projeto. Esta característica pode impactar diretamente na qualidade do circuito final, uma vez que a utilização de diferentes estilos lógicos pode levar a circuitos mais eficientes em termos de área, potência e atraso (BHATTACHARYA, 2002). Como exemplo, pode-se citar os estilos lógicos PTL (*Pass Transistor Logic*) e CMOS não-complementar série/paralelo como alternativas a serem utilizadas para a composição do circuito.

Neste contexto, este trabalho apresenta um gerador automático de redes de transistores capaz de fornecer diferentes tipos de redes em diversos estilos lógicos. Para comparar as redes geradas, algumas técnicas de estimativa são empregadas. Comparações são realizadas sobre conjuntos distintos de funções Booleanas, demonstrando as vantagens da utilização de lógicas alternativas em relação ao difundido padrão CMOS.