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

THIAGO ROSA FIGUEIRÓ

# Multiple objective technology independent logic synthesis for multiple output functions through AIG functional composition

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

PhD Prof. André Inácio Reis
Advisor

PhD Prof. Renato Perez Ribas
Co-advisor

Porto Alegre, October 2010.

**CIP – CATALOGAÇÃO NA PUBLICAÇÃO**

Figueiró, Thiago Rosa

Multiple objective technology independent logic synthesis for multiple output functions through AIG functional composition / Thiago Rosa Figueiró – Porto Alegre: Programa de Pós-Graduação em Microeletrônica, 2010.

77 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica. Porto Alegre, BR – RS, 2010. Orientador: André Inácio Reis; Co-orientador: Renato Perez Ribas.

1. Logic Synthesis. 2. And-Inverter Graph. 3. Design Automation I. Reis, André Inácio. II. Ribas, Renato Perez. III. Multiple objective technology independent logic synthesis for multiple output functions through AIG functional composition.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PGMicro: Prof. Ricardo da Luz Reis
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

# ACKNOWLEDGEMENTS

I would like to thank my advisor, André Inácio Reis for accepting me as his student even knowing that I could only provide a part-time dedication to this work. Moreover, I would like to acknowledge his support in providing and discussing ideas that are the core of this thesis. Moreover, I would like to thank him and my co-advisor, Renato Perez Ribas, for the discussions about career and future expectations. It is not easy to advise a part-time student and the results obtained by this work shows that it is possible.

This work was developed while working at Nangate. The support and understanding from my colleagues were essential so I could obtain my Master of Science degree.

During this project, although I was not a member of the LogiCS Lab, I spent several weeks working there and I'm very thankful for the support and friendship received from the entire team. I would like to specially thank Mayler Martins for his contribution in this work and Alex Goulart, who helped a lot in making things happen in the bureaucracy field.

Moreover, I want to thank my mother Cristina, who gave all the support I need in several hard times. Also, I would like to thank my brothers Rodrigo and Otávio, who understood my absence in several moments. Specially, I would like to acknowledge the support and caring received from my wife Nivea, whose love made this work possible.

# TABLE OF CONTENTS

LIST OF ABREVIATIONS ........ 7

LIST OF FIGURES........ 8

LIST OF TABLES ........ 10

ABSTRACT ........ 12

RESUMO ........ 13

1 INTRODUCTION ........ 14

2 BACKGROUND ........ 17

2.1 Basic Concepts ........ 17

2.1.1 Boolean Equation........ 17

2.1.2 Cofactors and Cube-cofactors ........ 18

2.1.3 Function Order ........ 18

2.1.4 Unateness........ 19

2.1.5 Read-once Functions ........ 19

2.1.6 Truth Tables........ 19

2.1.7 Binary Decision Diagram........ 20

2.1.8 And-Inverter Graph ........ 22

2.2 Logic Synthesis Flow ........ 23

2.3 Technology Independent Logic Synthesis Flow ........ 24

2.3.1 Equation Factorization........ 25

2.3.2 Using Boolean Decision Diagrams........ 25

2.3.3 Equation Composition ........ 25
2.3.4 Three-level And-Not Networks ........ 26
2.3.5 Using And-Inverter Graph ........ 27
2.3.6 Comparing Methods ........ 28
2.4 Cost Estimation ........ 29
2.4.1 Optimization Targeting Area ........ 29
2.4.2 Optimization Targeting Delay ........ 30
2.4.3 Optimizing Multiple Targets ........ 32
2.5 Related Tools ........ 33
2.5.1 ABC Tool ........ 33
3 PROPOSED METHOD ........ 34
3.1 Building the Allowed Functions ........ 35
3.2 Create Single Variable Functions ........ 38
3.3 Combining and Evaluating Functions ........ 39
3.4 Cost Functions and Implications ........ 42
3.5 Multiple Outputs ........ 43
3.6 Different Costs for Inputs ........ 44
3.7 Disjoint Approach ........ 45
3.8 Algorithm Complexity ........ 47
4 EXAMPLES ........ 48
4.1 Main algorithm ........ 48
4.2 Logic sharing example ........ 49
4.3 Multiple outputs example ........ 51
4.4 Different input weights example ........ 52
5 RESULTS ........ 54
5.1 Main Algorithm ........ 54
5.2 Multiple Outputs ........ 55
5.3 Secondary Criterion ........ 56

5.4 Different Costs for Inputs ..... 57

5.5 Disjoint Approach ..... 58

6 CONCLUSION ..... 59

REFERENCES ..... 62

APPENDIX SÍNTESE LÓGICA INDEPENDENTE DE TECNOLOGIA VISANDO MÚLTIPLOS OBJETIVOS, APLICADA A FUNÇÕES DE MÚLTIPLAS SAÍDAS, EMPREGANDO COMPOSIÇÃO FUNCIONAL DE AIGS ..... 65

ANNEX DESCRIPTION OF THE CLASSES AND CLASS DIAGRAM OF THE PROPOSED METHOD ..... 71

Function_Composer class ..... 71

AIG_Manager class ..... 72

AIG_Node class ..... 73

Truth_Table class ..... 74

Parse_Tree class ..... 74

Bucket class ..... 75

Bucket_Element class ..... 76

Function class ..... 76

Equation_Function class ..... 76

AIG_Function class ..... 77

# LIST OF ABREVIATIONS

| | |
|---|---|
| AIG | And-Inverter Graph |
| AND | Logic operation “and” |
| BDD | Binary Decision Diagram |
| DAG | Directed Acyclic Graph |
| FRAIG | Functional Reduced And-Inverter Graph |
| NAND | Negated logic operation “and” |
| NOR | Negated logic operation “or” |
| NOT | Inverted (Negated) Logic |
| OR | Logic operation “or” |
| POS | Product of Sums |
| ROBDD | Reduced Ordered Binary Decision Diagram |
| RTL | Register Transfer Level |
| SOP | Sum of Products |
| TANT | Three-level And-Not Networks |
| TT | Truth Table |
| XNOR | Negated exclusive-or operation |
| XOR | Exclusive-or operation |

# LIST OF FIGURES

Figure 2.1. BDD representing the function in Table 2.3. .............................................. 21
Figure 2.2. BDD representing the same function than BDD from Figure 2.1, but with a different variable ordering. ................................................................................ 21
Figure 2.3. ROBDD representing the same function than (a) BDD from Figure 2.1. and (b) BDD from Figure 2.2. ................................................................................ 22
Figure 2.4. Sample of AIG for the function represented by the equation F=A*!B*C. .. 22
Figure 2.5. Sample of AIG representing the same function but described as (a) F= A*B+C and (b) F=(A+C)*(B+C). ........................................................................ 23
Figure 2.6. Diagram representing the logic synthesis flow and its inputs and outputs. . 23
Figure 2.7. Pseudo-code for the Equation Composition algorithm. (REIS, 2009)......... 26
Figure 2.8. Pseudo-code of the AIGs construction by the FRAIG method (MISHCHENKO 2005). ................................................................................ 28
Figure 2.9. Small circuit showing two paths from input C to output Q, represented as Path 1 (dashes) and Path 2 (dots)................................................................................ 30
Figure 2.10.  A circuit long path from G to Q and a possible splitting of this circuit in three parts, (a), (b) and (c). ................................................................................ 31
Figure 2.11.  Same circuit path presented in Figure 2.10, including the hypothetical cost to the gates. ................................................................................ 31
Figure 2.12. Same circuit presented in Figure 2.9 presenting costs in the inputs and gates. ................................................................................ 32
Figure 3.1. Diagram presenting how the AND and OR operations are applied to the functions during composition. ................................................................................ 34
Figure 3.2. The (a) AND operation and the (b) OR operation over two AIGs, generating a new one. ................................................................................ 35
Figure 3.3. The pseudo-code of the algorithm. ........................................................... 35
Figure 3.4. The pseudo-code of the build allowed sub-functions method. .................... 36
Figure 3.5. The vector with all distinct cofactors and cube-cofactors for the target function f=A*B+C. ................................................................................ 37
Figure 3.6. The complete vector with all allowed sub-function. .................................. 38
Figure 3.7.  Diagram presenting how the AND and OR operations are applied to function representations during composition. The numbers in brackets indicates the algorithm step order. ................................................................................ 40
Figure 3.8. The pseudo-code of the combining sub-functions method. ........................ 40
Figure 3.9. (a) An AIG resulting for the function from the truth table presented in table 3.10 and (b) another AIG for the same function, presenting the same number of nodes but a smaller graph depth................................................................................ 43
Figure 3.10. (a) An AIG resulting for the function from the truth table presented in Table 3.11 without considering the inputs costs and (b) another AIG for the same

function, presenting the same number of nodes but a smaller graph depth as it considers the costs in Table 3.12. .......... 45
Figure 4.1. The AIG built by running ABC followed by FRAIG, for the equation (4.1). .......... 49
Figure 4.2. The AIG built by the proposed method for the equation (4.1). .......... 49
Figure 4.3. The AIG built by running ABC followed by FRAIG, for the equation (4.2). .......... 50
Figure 4.4. The AIG built by running the proposed method for Equation 4.2. Notice the circled node which presents fanout larger than one. .......... 51
Figure 4.5. The AIGs built by running the proposed method for (a) Equation 4.3 and (b) Equation 4.4, one at a time. .......... 51
Figure 4.6. The AIGs built by running the proposed method for Equation 4.3 and Equation 4.4, both at the same time in multiple outputs mode. .......... 52
Figure 4.7. The AIGs built by running the proposed method for the function represented by Equation 4.5 and different input costs presented in Table 4.3. (a) for cost 1, (b) for cost 2, (c) for cost 3 and (d) for cost 4. .......... 53
Figure 5.1. The distribution of number of nodes gained by the proposed method when compared to the ABC + FRAIG method, when applied to the input functions from set A. Negative values represent advantage to ABC + FRAIG, while positive values represent advantage to the proposed method. .......... 55
Figure 5.2. The impact of grouping functions in order to perform the multiple outputs algorithm in terms of average number of nodes and in terms of average logical depth decrease, when applied to the input functions from set A. .......... 56

# LIST OF TABLES

Table 2.1: Sample of truth table for the XOR function ................................................ 19
Table 2.2: Relation between the number of inputs, number of bits and number of integers required to store this information in both 32 and 64 bits architectures. ........... 20
Table 2.3: Truth table for Q=A*B+C equation. .......................................................... 20
Table 2.4: Comparison between properties of the discussed methods. ......................... 29
Table 2.5: The Logical Depth (Cost) of each path in the subcircuit presented in Figure 2.12. ............................................................................................................... 32
Table 3.1: Truth table for the equation F = (A+C)*(B+C). .......................................... 36
Table 3.2: Cofactors of the target function 01010111. ................................................. 37
Table 3.3: Cube-cofactors of the cofactors of the target function Q = A*B+C. ............ 37
Table 3.4: AND operations applied over the cofactors and cube-cofactors resulting of previous steps. ............................................................................................... 38
Table 3.5: OR operations applied over the cofactors and cube-cofactors resulting of previous steps. ............................................................................................... 38
Table 3.6: Truth tables of the variable functions and their inverted values. ................. 39
Table 3.7: OR (+) and AND(*) operations applied over the one variable functions. .... 41
Table 3.8: OR operations applied over the cofactors and cube-cofactors resulting of previous steps. ............................................................................................... 41
Table 3.9: OR operations applied (+) and AND(*) operations applied over the one variable functions (o nodes functions) with the 1 node functions. ................................ 41
Table 3.10: Truth table for the equation Q = A*B+C+D. ............................................ 42
Table 3.11: Truth table for the equation Q = !A*B*!C*!D+!A*C*D. .......................... 44
Table 3.12: Costs for the inputs in the proposed example............................................ 44
Table 4.1: Truth table for the equation Q=((!B*!C*D)+(B*!C*!D)+(!A*C*D)+(A*!B*!D)) ...................................... 48
Table 4.2: Truth table for the equation Q= !B*!C*D+!A*(B*C*D+!D*(B*!C+!B*C)) 50
Table 4.3: Costs used as inputs costs to the AIG construction algorithm. .................... 52
Table 5.1: Comparison of And-Inverter Graphs generated by ABC + FRAIG and the proposed method for set A of input functions. ............................................................ 54
Table 5.2: Evaluation of multiple outputs result for groups of 2 or 4 functions compared to single output as well ......................................................................................... 55
Table 5.3: Comparison of And-Inverter Graphs generated by Equation Composition + FRAIG and the proposed ............................................................................................ 56
Table 5.4: Comparison of And-Inverter Graphs generated by Equation Composition + FRAIG and the proposed ............................................................................................ 57
Table 5.5: Inputs Cost vectors used during this test. .................................................... 57
Table 5.6: Comparison of And-Inverter Graphs generated by the proposed method either ignoring or not the inputs cost ........................................................................ 57

Table 5.7: Comparison of And-Inverter Graphs generated by the proposed method in both regular and disjoint efforts........................................................................................ 58

# ABSTRACT

The use of design automation tools has allowed complex projects to reach feasible time-to-market and cost parameters. In this context, logic synthesis is a critical procedure in the design flow. The technology independent step (part of the logic synthesis which is performed regardless any physical property) is traditionally performed over equations. The development of new multi-level optimization algorithms has recently shifted towards the use of And-Inverter-Graphs (AIGs). The number of nodes and the graphs depth in AIGs present better correlation with resulting circuit area and delay than any characteristic of other representations. In this work, a technology independent synthesis algorithm that works on top of an AIG data structure is proposed. A novel approach for AIG construction, based on a new synthesis paradigm called functional composition, is introduced. This approach consists in building the final AIG by associating simpler AIGs, in a bottom-up approach. The method controls, during the graphs construction, the characteristics of final and intermediate graphs by applying a cost function as a way to evaluate the quality of those AIGs. The goal is to minimize the number of nodes and the depth of the final AIG. This multi-objective synthesis algorithm has presented interesting features and advantages when compared to traditional approaches. Moreover, this work presents a method for AIGs construction for multiple output functions, which enhances structural sharing, improving the resulting circuit. Results have shown an improvement of around 5% in number of nodes when compared to ABC tool.

**Keywords:** Logic Synthesis, And-Inverter Graph, Design Automation, CAD, Digital Circuits, Logic Gates, Design Flow, Technology Mapping

**Síntese lógica independente de tecnologia visando múltiplos objetivos, aplicada a funções de múltiplas saídas, empregando composição funcional de AIGs**

# RESUMO

O emprego de ferramentas de automação de projetos de circuitos integrados permitiu que projetos complexos atingissem *time-to-market* e custos de produção factíveis. Neste contexto, o processo de síntese lógica é uma etapa fundamental no fluxo de projeto. O passo independente de tecnologia (parte do processo de síntese lógica, que é realizada sem considerar características físicas) é tradicionalmente realizado sobre equações. O desenvolvimento de novos algoritmos de otimização multi-nível recentemente migrou para o emprego de *And-Inverter Graphs* (AIGs). O número de nodos e a altura de um grafo apresentam melhor correlação com os resultados em área e atraso de um circuito, se comparados com as características de outras formas de representação. Neste trabalho, um algoritmo de síntese lógica independente de tecnologia, que funciona sobre uma estrutura de AIGs, é proposto. Uma nova abordagem para a construção de AIGs, baseada no novo paradigma de síntese chamado de composição funcional, é apresentado. Esta abordagem consiste em construir o AIG final através da associação de AIGs mais simples, em uma abordagem *bottom-up*. Durante a construção do grafo, o método controla as características dos grafos intermediários e finais, a partir da aplicação de uma função de custo, como forma de avaliação da qualidade desses AIGs. O objetivo é a minimização do número de nodos e da altura do AIG final. Este algoritmo de síntese lógica multi-objetivo apresenta características interessantes e vantagens quando comparado com abordagens tradicionais. Além disso, este trabalho apresenta a síntese de funções com múltiplas saídas em AIGs, o que melhora a característica de compartilhamento de estruturas, melhorando o circuito resultante. Resultados mostraram a melhora em torno de 5% em número de nodos, quando comparados com os resultados obtidos com a ferramenta ABC.

**Palavras-chave:** Síntese lógica, And-Inverter Graph, Automação de Projetos, CAD, Circuitos Digitais, Portas Lógicas, Fluxo de Projeto, Mapeamento Tecnológico.

# 1 INTRODUCTION

The advances in the integration scale of electronic devices have increased drastically the complexity of developing an integrated circuit (IC). Therefore, Electronic Design Automation (EDA) tools and an adequate project approach are essential to enable engineers to meet project requirements in a feasible time and without increasing the project cost. One of the most frequently adopted approaches for developing Application Specific Integrated Circuits (ASICs) is the standard cell project flow. The standard cell flow is based on cell libraries, consisting on having a specific design for a predefined set of logic gates. This approach leads to reduce the design time because of the reuse of previous made layouts in different circuits and of different parts of the same circuit.

A standard cell design flow consists of a sequence of steps starting from an abstract description of the circuit and resulting in the entire circuit, with all its elements properly specified, placed and routed. A standard cell flow may be divided into two main parts: logic synthesis and physical synthesis. The logic synthesis is responsible for optimizing the original description of the circuit and selecting the most appropriate cells to describe it. The physical synthesis is responsible for placing and routing these cells in the circuit area.

Traditionally, logic synthesis is usually performed in two steps: one performed over Boolean equations (regardless any physical property) and another where the resulting logic is mapped into a cell library or any other physical implementation. The first step is known as *technology independent step* and the second one is called *technology dependent step*.

During many years, the technology independent step was performed by equation factoring. Although it is a core procedure in logic synthesis, the only known optimality result for factoring (until 1996) was the one presented by Lawler in 1964, according to (HACHTEL, 1996). Heuristic techniques have been proposed for factoring and they have achieved high commercial success. This includes the quick_factor (QF) and good_factor (GF) algorithms available in the SIS tool (SENTOVICH, 1992). SIS (and others similar algorithms) is composed of several logic operations such as Decomposition Extraction, Factoring, Substitution and Elimination (HATCHTEL, 1996).

Most of the proposed factoring algorithms for technology independent logic synthesis take as input a sum-of-products (SOP) or a product-of-sums (POS). As SOP/POS forms are completely specified, the *don't cares* are not treated during the factoring but while generating the SOP/POS. Thus, the whole process is not completely optimized (not exact). However, algorithms that start from functional descriptions of the

functions and are able to handle *don't cares* are usually too slow to complete their execution for all functions of 4 variables (REIS, 2009).

Recently, factoring methods that produce exact results for functions that may be expressed without repeating literals (read-once factored forms) have been proposed (GOLUMBIC, 2001) and improved (GOLUMBIC, 2008). However, the proposed IROF algorithm is restrict to read-once functions. The Xfactor algorithm (GOLUMBIC, 1999) is exact for read-once forms and produces good heuristic solutions for functions not included in this category.

Recently, a novel approach based on function composition was proposed by (REIS, 2009). It works in a bottom-up way combining equations in order to generate the equation that implements the target function. The great achievement of this method is to provide good solutions in reasonable time, allowing optimizing multiple goals.

Other approach is the use of Binary Decision Diagrams (BDDs) for performing technology independent optimizations. The advantages of this approach rely mainly in the existence of well known algorithms for handling BDDs and the fact that BDDs do not depend of a previous equation description of the function. However, mapping circuits over a BDD is far more difficult than mapping over other Direct Acyclic Graphs (DAGs).

Three-level And-Not Networks (TANTs) were also considered as an alternative to equation factorization to perform technology independent optimization. The main advantage of TANT is the possibility of logic sharing and the fact that its description has as start point a function description. However, TANTs are not appropriate for implementing multi-objective optimizations (which consist in targeting more than one circuit characteristic at the same time, such as circuit area, delay propagation, etc.) and the characteristics of the TANT's graph are not strongly reflected in the resulting mapped circuit.

The factoring algorithms presented in (GOLUMBIC, 1999; REIS, 2009) do not include sub-expression reuse. The reuse of sub-expressions makes multi level representations, like TANT networks (LEE, 1978), more competitive than two-level expressions (MISCHENKO, 2003). The sub-expression reuse properties of multi level representations lead to another interesting use of these methods, which is multiple output synthesis. Supporting multiple functions allows the logic sharing of sub-expressions presented in more than one function, enhancing the circuit timing and consumption by reducing circuit area.

The development of new multi-level optimization algorithms has recently shifted towards the use of AIGs (MISHCHENKO, 2006). The use of AIG nodes is justified as it is expected to produce better correlation with final area and delay once the circuit has been mapped to a target technology (MISHCHENKO, 2006). This advantage, when comparing to using equations, comes from: (1) the fact that AIGs are multi-level representations, which allows sharing of nodes; and (2) the AIG node is a simple structure, which keeps correlation with area as all nodes have homogeneous simple granularity. However, a recent work (JOSWIAK, 2008) has shown that for small circuits there is still room for area gains with respect to AIG based tools. In this sense, there is a need for a Boolean algorithm for AIG rewriting.

The title of this work, “Multiple objective technology independent logic synthesis for multiple output functions through AIG functional composition”, may be explained as follows:

- Multiple objective: consider a cost function with more than one parameter, in order to improve more than one characteristic that reflects improvements in the resulting circuit. In this work, number of nodes followed by graph height;
- Technology independent logic synthesis: step on the logic synthesis flow which is performed regardless any physical property;
- For multiple output functions: able to support several functions at a time, enhancing logic sharing.
- Through AIG functional composition: build functions as And-Inverter Graphs in a bottom-up approach.

The main purpose of this work is to propose a method to perform algorithmic logic synthesis using AIGs (due to the characteristics previously mentioned) associated with the new synthesis paradigm called *functional composition* (REIS, 2009). The approach proposed herein consists of constructing AIGs from association of simpler ones, in a bottom-up approach. The method controls, during the graphs construction, the characteristics of final and intermediate graphs by applying a cost function as a way to evaluate the quality of those AIGs (the cost function may reflect only the number of nodes, graph height or an association of these two criteria). This work also discuss the possibility of considering initial costs for the function variables, which may be used to represent the different arrival time of signals in a circuit and, therefore, that a given input must be favored in detriment of others. Moreover, an approach for handling multiple output functions during the function composition is proposed, presenting good results, especially when compared to handling each input separately. A post processing algorithm for duplicating logic in case of extremely large node fanouts makes the circuits resulting from this approach feasible.

Chapter 2 presents the basic concepts regarding logic synthesis and a short comparison among the most frequently adopted approaches, besides discussing briefly cost estimation fundamentals. Chapter 3 presents the proposed algorithm for AIG composition, describing each step of the main algorithm and detailing the characteristics for special situations such as optimizing multiple aspects of the resulting circuit and optimizing multiple output circuits. Chapter 4 presents examples of AIGs constructed by the proposed method and comparisons to other methods. The results are presented and discussed in Chapter 5. This chapter also compares the results from the proposed method with the results from the function composition method applied to equations (instead of AIGs). Moreover, the results from the proposed method are compared to the ones from the approach of using ABC (running equation factoring, using “good_factor” from SIS, and building the AIGs using the FRAIG algorithm, which is also embedded in ABC). Chapter 6 presents the conclusion of this work and discusses the next steps for this research. In the annex the class diagram and a short description of all classes are presented.

# 2 BACKGROUND

This chapter presents the basic concepts and the state of the art of the subjects of interest in this work. Section 2.1 discusses Boolean equations, the concept of cofactors, cube-cofactors and read-once functions. Moreover, it presents truth tables, binary decision diagrams (BDDs) and and-inverter graphs (AIGs). This review is important to understand the basis of the state of the art algorithms and the proposed method's characteristics.

Sections 2.2 and 2.3 present general aspects of the logical synthesis flow and detail the state of the art of the most frequently approaches adopted to perform the technology independent synthesis step. Moreover, in the end of Section 2.3, a comparison between these methods is performed. This comparison is part of the motivation of this work since any of the presented methods is able to provide all the desirable qualities. Finally, Section 2.4 presents the concept of cost estimation for a solution and the approaches for targeting area, delay or both. Cost estimation is an important concept since this work aims to improve multiple aspects of the resulting circuit.

## 2.1 Basic Concepts

### 2.1.1 Boolean Equation

A Boolean function describes how to determine a Boolean output based on some logical calculation performed over Boolean inputs. An equation is one representation of a function, which may also be described as a Binary Decision Diagram (BDD) or as a Truth Table (TT), for instance. Every representation of a function may be classified as canonical or non-canonical. A representation is said to be canonical if every function will always be described exactly in the same way. Examples of canonical representations are BDDs and TTs (as long as the variables ordering are the same). Usually, equations are non-canonical representations of a function; therefore, the same function may be described by different equations. For instance, equations (2.1) and (2.2) represent exactly the same function. However, there is a way to represent an equation in a canonical way, which is the SOP representation. An equation is composed of literals and operators. A literal is an instance of a variable (positive literal, for instance "A") or its complement (negative literal, for instance "!A"). Operators are AND ("*"), OR ("+") and NOT ("!").

$$F=A*B+C \quad (2.1)$$

$$F=(A+C)*(B+C) \quad (2.2)$$

### 2.1.2 Cofactors and Cube-cofactors

A Shannon Decomposition is a method to represent any Boolean function as the sum of two sub-functions of the original one. A cofactor is a sub element of a Shannon Decomposition generated by either setting the value of a given variable to "0" or to "1", When a cofactor is generated for a function *F* by setting a variable *v* to "0", it is called the negative cofactor of the function *F* with respect to variable *v*.

For instance, equation (2.3) presents a function F represented by it's two sub elements, one related to x and the other one related to !x. There are the positive and negative cofactors of the F function in x, respectively.

$$F=x*F_{x} +!x*F_{!x} \quad (2.3)$$

A cofactor with respect to a given variable is obtained by setting the variable to one (positive cofactor) or to zero (negative cofactor), eliminating then the variable from the function. For example, consider the function (2.4).

$$F=(A*C)+(B*C)+(!A*B*D) \quad (2.4)$$

The positive cofactor with respect to the variable A (which is generated by setting A to one) is:

$$F_{(A=1)}= C+(B*C) \quad (2.5)$$

While the negative cofactor with respect to the variable A (which is generated by setting A to zero) is:

$$F_{(A=0)}= (B*C)+(B*D) \quad (2.6)$$

A cube-cofactor is obtained when setting more than one variable to a fixed logic value. In other words, a cube-cofactor is a cofactor from a cofactor. The cube-cofactor of the function presented in (2.4) with respect to A=0 and B=1 is presented in (2.7)

$$F_{(A=0, B=1)}= C+D \quad (2.7)$$

### 2.1.3 Function Order

Two Boolean functions may be compared and classified according to their relative order. Considering $G = F1 + F2$:

- $F_1$ **is larger than** $F_2$ if $F_1 = G$ and $F_1 \neq F_2$
- $F_1$ **is smaller than** if $F2$ if $F2 = G$ and $F_1 \neq F_2$
- $F_1$ **not comparable** to $F_2$ if $F_1 \neq F_2 \neq G$

### 2.1.4 Unateness

Boolean function variables may be classified according to their unateness. A function $F$ is said to be positive (negative) unate in a variable $v$ if the positive cofactor of $v$ in $F$ is larger (smaller) than the negative cofactor of $v$ in $F$. If the positive and negative cofactors are not-comparable, the function $F$ is said to be binate in $v$. Therefore:

- $F$ **is positive unate in** $v$ if $F_{v=1}$ is larger than $F_{v=0}$
- $F$ **is negative unate in** $v$ if $F_{v=1}$ is smaller than $F_{v=0}$
- $F$ **is binate in** $v$ if $F_{v=1}$ is not comparable to $F_{v=0}$

### 2.1.5 Read-once Functions

A function $F$ is called read-once if it can be represented as a factored form in which each variable appears only once. In other words, a function is said to be read-once if there is a factored form of it that do not present any repeated literal.

If a function $F$ can be represented through a read-once formula, all the partial sub-equations in the formula correspond to functions that are cube cofactors of $F$. As each variable appears as a single literal, they can all be independently set to non-controlling values, which makes only one literal disappear at a time. This way, any sub-equation (or sub-set) of $F$ can be obtained by assigning non-controlling values to the variables to be eliminated.

### 2.1.6 Truth Tables

A truth table (TT) is another way to represent Boolean functions. It consists in a table containing columns for all inputs and for the output of the function. Each line of the table consists in one combination of values for the inputs. For instance, the XOR function of two inputs is presented in Table 2.1.

Table 2.1: Sample of truth table for the XOR function

| *Input 1* | *Input 2* | *Output* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

If two TTs present the same inputs in the same order, they will only vary on the output column. Therefore, a representation of a TT must only present the data of this column and the input names and ordering. The output column of the TT may be seen as a vector of Boolean values (for the example in Table 2.1, the output vector would be [0,1,1,0]). The vector may also be described as an integer (again in the example in Table 2.1., the integer would be 0110 in binary representation, or 6, in decimal

representation). The number of bits of an integer may vary from architecture to architecture, but usually they are limited to 32 or 64 bits. If the number of lines in the TT (and therefore the number of bits in the vector or the number of bits in the integer) is larger than the number of bits in a single integer, it is possible to use a vector of integers to store the TT data. Table 2.2 presents the relation between the number of inputs, number of bits and number of integers required to store this information in both 32 and 64 bits architectures.

Table 2.2: Relation between the number of inputs, number of bits and number of integers required to store this information in both 32 and 64 bits architectures.

| *Inputs* | *Bits* | *Integers (32 Bits)* | *Integers (64 Bits)* |
|---|---|---|---|
| 1 | 2 | 1 | 1 |
| 2 | 4 | 1 | 1 |
| 3 | 8 | 1 | 1 |
| 4 | 16 | 1 | 1 |
| 5 | 32 | 1 | 1 |
| 6 | 64 | 2 | 1 |
| 7 | 128 | 4 | 2 |
| 8 | 256 | 8 | 4 |
| 9 | 512 | 16 | 8 |
| 10 | 1024 | 32 | 16 |
| n | $2^n$ | $2^n / 32$ | $2^n / 64$ |

### 2.1.7 Binary Decision Diagram

A Binary Decision Diagram (BDD) is a directed acyclic graph (DAG) which represents a Boolean function. The nodes contain the variables while the edges represent the values assumed by the variables. If no reduction rules are applied, the BDD is actually a tree, presenting the output node of the function it implements as the root and Boolean values ‘1’ and ‘0’ at the leaves. Figure 2.1 presents the BDD and the TT for the function represented by the equation Q=A*B+C.

Table 2.3: Truth table for Q=A*B+C equation.

| *A* | *B* | *C* | *Q* |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

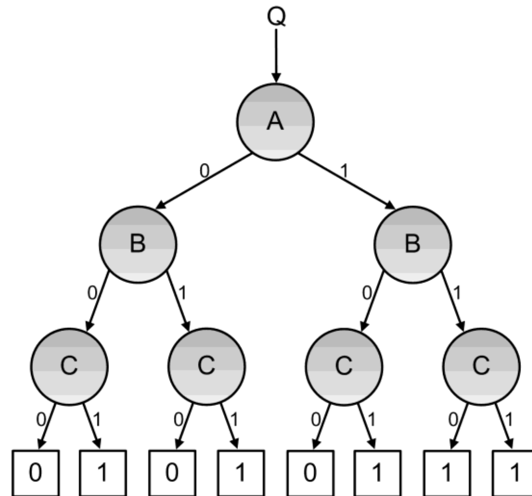



Figure 2.1. BDD representing the function in Table 2.3.

Considering the BDD without any reduction, it may be considered canonical if and only if the ordering of the variables is specified (for instance, if the variables are ordered lexicographically). For a different variable ordering (for instance, in Figure 2.2 the order is C, A and B, instead of A, B and C in Figure 2.1), the BDDs will differ.

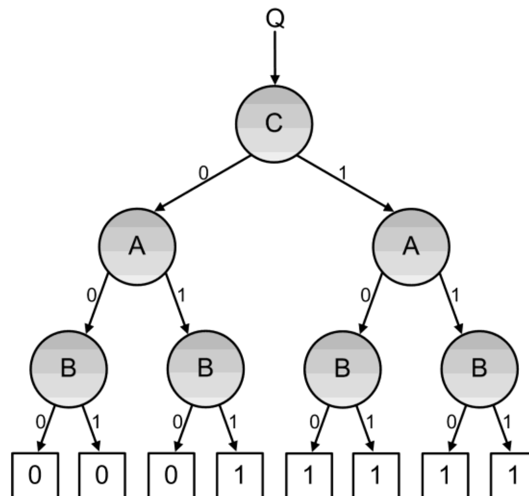
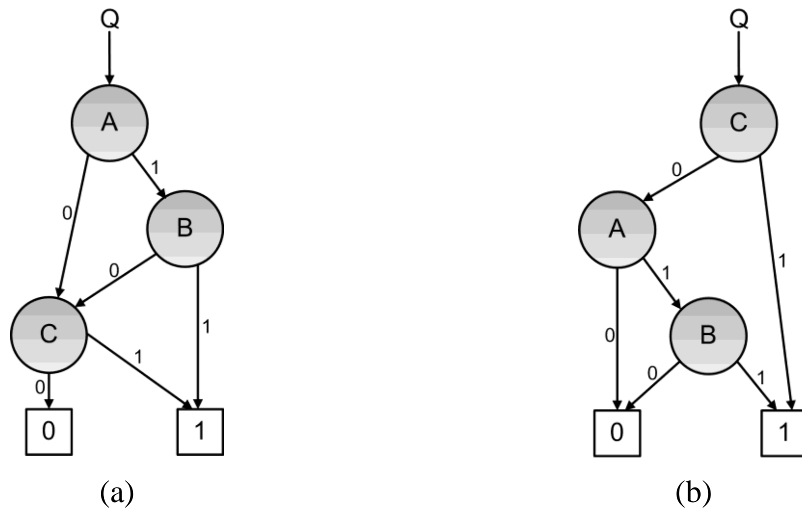



Figure 2.2. BDD representing the same function than BDD from Figure 2.1, but with a different variable ordering.

The size of a BDD grows exponentially with the number of variables. Moreover, the time for building a BDD is also exponentially proportional to its number of variables. Therefore, several different techniques to reduce the size of a BDD were proposed. One of the most popular approaches is called ROBDD (which stands for Reduced Ordered Binary Decision Diagram). The ROBDD is a BDD with some specific improvements in building time (which does not increase the time for building the BDD). The most significant improvement is that if there are two identical nodes, they are collapsed.



(a) (b)

Figure 2.3. ROBDD representing the same function than (a) BDD from Figure 2.1. and (b) BDD from Figure 2.2.

The advantage of using ROBDD instead of simple BDDs is the gain in size (number of nodes and complexity) without a significant penalty in building time. Moreover, it maintains the characteristic of being canonical, as long as the variable ordering is the same (as simple BDDs).

### 2.1.8 And-Inverter Graph

And-Inverter Graph (AIG) is another way to represent a Boolean function. An AIG is a directed acyclic graph (DAG) which is composed exclusively of two inputs AND gates and inverters. The inverters are usually represented as a special flag on the graph's edge and therefore all nodes on the graph represent two input ANDs. Figure 2.4 presents one AIG for the equation F=A*!B*C.

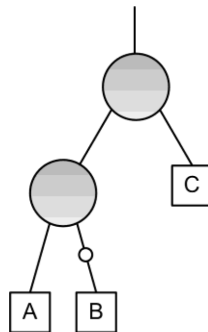



Figure 2.4. Sample of AIG for the function represented by the equation F=A*!B*C.

As AIGs are not canonical, different AIGs may represent the same Boolean function. Figure 2.5 presents three different AIGs for the same function (the output of the TT of this function is 01010111).

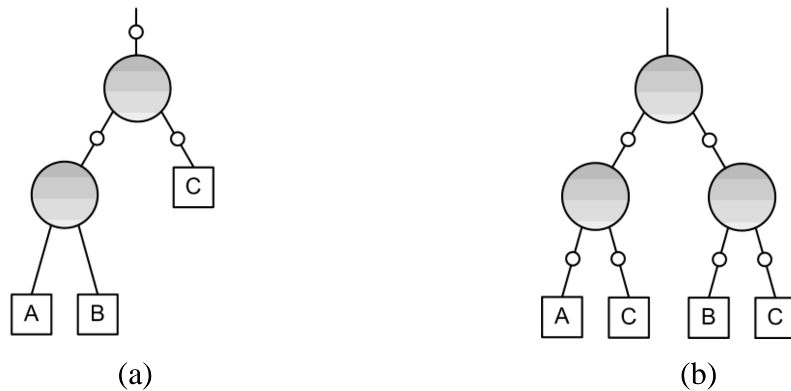



Figure 2.5. Sample of AIG representing the same function but described as (a) F= A*B+C and (b) F=(A+C)*(B+C).

## 2.2 Logic Synthesis Flow

Logic synthesis is the process to generate a design implementation of a circuit from its abstract description (typically a structural or a RTL – register transfer level - description). Figure 2.6 presents a diagram representation of the logic synthesis steps and inputs.

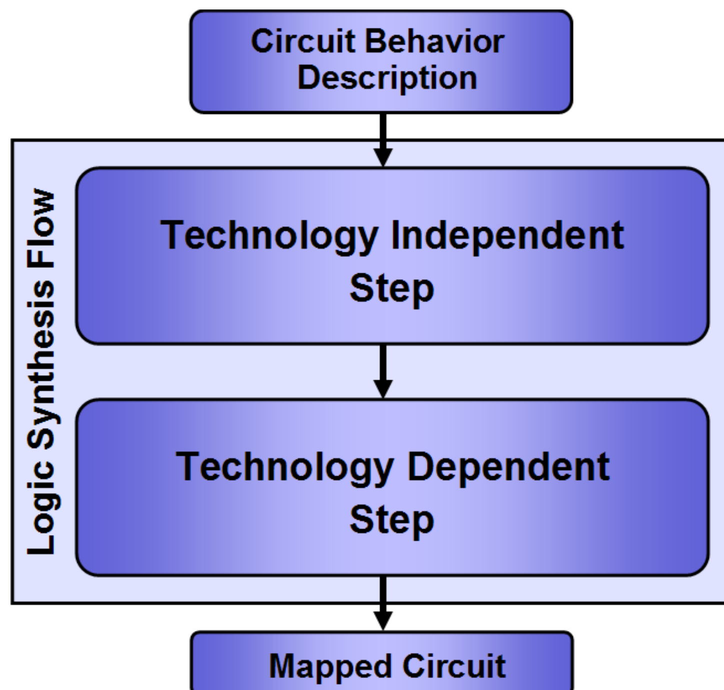



Figure 2.6. Diagram representing the logic synthesis flow and its inputs and outputs.

The abstract description must inform mainly the logic behavior of the circuit to be synthesized and, therefore, must describe at least the set of inputs and outputs and the values of the outputs for all combination of the inputs (for combinational circuits) or

must describe both the combination of the inputs and previous state information (for sequential circuits).

Considering only combinational circuits (in order to simplify the flow analysis), the logic synthesis is usually performed in two steps, one performed over Boolean equations (regardless any physical property) and another where the resulting logic is mapped into a physical cell library or other physical implementation.

The first step, called technology-independent step, may be either a two-level approach or a multi-level approach. The two-level approach consists of representing a function as a SOP, which means that the first level contains product terms (AND) and the second level logic contains sum terms (OR). It may also be necessary to employ inversions (NOT) to some of the inputs of the product terms.

The multi-level approach is composed of several logic operations such as Decomposition, Extraction, Factoring, Substitution and Elimination (HACHTEL, 1996). These operations may be either explicitly performed (SENTOVICH, 1992) or implicitly performed by other methods such as And-Inverter Graph (AIG) rewriting, as in the ABC tool (BERKELEY 2010). The result of this technology-independent step is an improved abstract description of the circuit. The typical cost function during this optimization is the total literal count of the factored representation of the logic function, although recent works proposed different cost functions considering multiple goals.

The second step, called technology-dependent step, uses the improved abstract description provided by previous step to build a DAG and perform technology mapping over this graph. The result of this step (and consequently, of the logic synthesis flow) is the circuit described as a network of gates in a given technology.

While the technology-independent step may be performed in the same way either in an ASIC or in a FPGA flow, the technology-dependent step must consider specific restrictions for each flow and, therefore, must present specific characteristics for ASIC and for FPGA flows. ASIC flow restricts technology mapping to the available cells in the cell library, while FPGA flow presents restriction regarding the maximum number of inputs of a given cell and other resources restrictions.

Some of the most used techniques for the technology-independent step are discussed in the next section.

## 2.3 Technology Independent Logic Synthesis Flow

Technology independent synthesis consists in computing a representation of a given combinational circuit with optimized costs measured independently of the target technology. These costs may be related to the number of literals in an equation (e.g. number of literals and logic depth) or TANT (LEE 1978, PERKOWSKI 1990), BDD (YANG, 1999 – VEMURI, 2002) or AIG (KUEHLMANN, 2002, MISHCHENKO 2005, FIGUEIRO 2010) nodes (e.g. number of nodes and graph depth measured in nodes). In SIS (SENTOVICH 1992), the technology independent cost was based on literals. In more recent tools like ABC (BERKELEY 2010), the technology independent cost is based on AIG nodes.

The following sections discuss the most relevant algorithms types for performing technology independent synthesis.

### 2.3.1 Equation Factorization

Equation Factorization was the first approach used in the technology-independent step in the logic synthesis flow. It consists of decomposing the equation in smaller and relatively independent elements, called factors. Equation 2.4 presents a simple equation in SOP form while equation 2.5 presents the same function implemented as a factorized equation.

$$F=A*B+A*C \tag{2.4}$$

$$F=A*(B+C) \tag{2.5}$$

The factorized equation always implements the same function than the original equation, but it usually presents a reduced number of literals once it groups equivalent elements. This enhancement in the equation is also reflected in the resulting circuit since it allows logic sharing of some equivalent elements.

This approach applies a sequence of optimization steps, having the goal of removing redundant nodes, finding better logic boundaries, discovering shared logic, and simplifying the node representations.

The method of factoring equations present several drawbacks, such as operating over equations and optimizing the number of literals, while technology mappers (CONG 1994, KUKIMOTO 1998) operate over DAGs and usually consider other cost functions than number of literals.

### 2.3.2 Using Boolean Decision Diagrams

Although Equation Factorization may present good results in most AND-OR structures, it is not completely satisfactory when optimizing circuits which present some XOR behavior.

The BDD structure presents several characteristics that would be interesting to the technology independent optimization step (YANG, 1999 – VEMURI, 2002). These characteristics are:

- BDDs are canonical (when variables are ordered);
- BDDs allow logic sharing;
- Reductions on BDDs usually reflect reductions in circuit logic;
- Variable reordering on BDDs usually represent gain in final optimization.

The reason why BDDs are not widely used in the technology independent step in logic synthesis flow is that a BDD is not a good structure for technology mapping and it is not easy to convert the BDD to such structure.

### 2.3.3 Equation Composition

Equation Composition is a novel approach (REIS, 2009) based on the construction of equations by associating simpler sub-equations. The objective of this method is to provide an equation that represents a good description of a given input function.

Figure 2.7 presents a description in pseudo-code of the algorithm. The method consists in first determining a set of allowed sub-functions, which will be the functions that are considered as possible partial elements of the target function (line 5). The second step is to create equations for all functions considering only one variable (line 7). The algorithm continues by combining these equations two by two until composing the final equation (lines 10 to 14). In order to evaluate whether a generated equation implements or not an allowed function (and therefore must be considered in further compositions). A TT is used for each equation, which specifies in a canonical way the functions (it was also possible to use BDDs in this point, but at (REIS, 2009) only TTs were used).

```
1   vector<factoredForms> optimizeEquation()
2   {
3     if (target_function==constant)
4        return constant;
5     compute_allowed_subfunctions();
6     bool sf ← false; //solutions found
7     sf=create_literals();
8     if (sf)
9        return solutions;
10    else for (int i=2; i<maxLit; i++){
11       sf=fillBucket(i);
12       if (sf)
13          return solutions;
14    }
15    return "no solutions found";
16  }
```

Figure 2.7. Pseudo-code for the Equation Composition algorithm. (REIS, 2009).

This method is able to constructively control the characteristics of final and intermediate equations, allowing the adoption of secondary criteria other than the number of literals for optimization. The drawback of using Boolean equations in the optimization step is the difficulty of handling logic sharing.

### 2.3.4 Three-level And-Not Networks

A Three-level And-Not Network (TANT) is a network composed exclusively of ANDs and inverters (NOTs) whose topology present only three levels (LEE, 1978, PERKOWSKI, 1994). It was proven (SASAO, 1982) that by using three logic levels it is possible to represent in a small quantity of nodes nearly all Boolean functions. On the other hand, using only two levels (such as in SOP or POS approaches) it affects drastically the logic size (SASAO, 1982).

Therefore, TANT networks are able to represent logic functions in a reduced form and there are several algorithms that perform this reduction efficiently and with low computational effort (LEE 1978, VINK 1978, PERKOWSKI 1990). Moreover, in opposition to other approaches, TANT networks are able to properly handle logic sharing by connecting several elements of a higher level to the same element in the lower level, increasing its Fan-In to more than 1.

However, one limitation of the TANT networks is exactly the number of levels (only three levels are possible). Moreover, the number of nodes in the TANT network is not directly correlated to the final circuit size (different fan-in nodes represent different

area in the resulting circuit), which both reduce the benefit of the nodes reduction (once it can actually represent an area increase of the circuit) and make difficult to apply TANT networks in multiple goals optimizations.

### 2.3.5 Using And-Inverter Graph

AIGs are a multi-level logic representation, whose construction and size are proportional to the size of the circuit. The characteristics of the AIG are better correlated to the circuit characteristics than any other strategy mentioned in this chapter.

The best possible equation does not ensure the best AIG and, therefore, another AIG generated by a not-so-good equation may be the best AIG. This is the reason why recent logic synthesis applications are based in AIG and AIG optimization.

ABC (BERKELEY 2010) uses AIGs for optimizing the logic before performing any technology mapping. The AIG is first built and than AIG rewriting techniques are applied in order to reduce its number of nodes (MISHCHENKO 2006, BRUMMAYER 2006).

The rewriting procedure is a fast greedy algorithm that reduces the AIG size by iteratively selecting its subgraphs and replacing them with smaller pre-computed subgraphs, with the same functionality. The rewriting technique presented by (MISHCHENKO 2006) extends the work from (BJESSE 2004), by restricting rewriting to preserve the number of logic levels, using 4-feasible cuts instead of two-level subgraphs and by balancing AIGs using algebraic tree-height reduction.

It consists in enumerating all 4-feasible cuts and, for each cut, computing the Boolean function and search for its NPN-class in a hash-table. Fast manipulation of 4-variables functions is achieved by representing them as TT of 16 bits bit-strings. After this first reduction procedure, logic sharing between the new subgraphs and nodes already existent in the complete AIG is determined. In case of total or partial logic sharing, these new shared nodes are removed and the old subgraph is used instead. After trying all available subgraphs for the given node, the one that leads to the largest improvement at a node is used.

Another approach to improve AIGs is the Functional Reduced And-Inverter Graph (FRAIG) technique (MISHCHENKO 2005), which is also available in ABC and may be used to generate the first AIG before performing AIG rewriting procedures.

The FRAIG technique consists in a method for building AIGs from a factorized Boolean equation. Figure 2.8 present the pseudo-code for the FRAIG construction.

The FRAIG construction algorithm starts by using a structural hashing for ensuring that some equivalent nodes present the same behavior. This may be performed in one-level or in two-level modes, which consider input permutations - or even node permutations in two-level mode (KUEHLMANN, 2002) - and structural changes in the nodes. The second step is performed to detect functional equivalence by calling a SAT solver (it was used the MiniSat (EEN 2004)).

The resulting AIG is considerably reduced from normal AIGs without penalty performance, once the optimizations are performed in construction time. Moreover, the FRAIG gives the AIG a semi-canonical form, since most of the equivalences are detected and translated to the same AIG.

```
Aig_Node * OperationAnd( Aig_Man * p, Aig_Node * n1, Aig_Nodee * n2 )
{
   Aig_Node * res, * cand, * temp; Aig_NodeArray * class;
   /*** trivial cases ***/
   if ( n1 == n2 ) return n1;
   if ( n1 == NOT(n2) ) return 0;
   if ( n1 == const ) return 0 or n2;
   if ( n2 == const ) return 0 or n1;
   if ( n1 < n2 ) { /*** swap the arguments ***/
      temp = n1; n1 = n2; n2 = temp;
   }
   /*** one level structural hashing ***/
   res = HashTableLookup( p->pTableStructure, n1, n2 );
   if ( res ) return res;
   res = CreateNode( p, n1, n2 );
   HashTableAdd( p->pTableStructure, res );

   if ( p->FlagUseOneLevelHashing ) return res;
   /*** functional reduction ***/
   class = HashTableLookup( p->pTableSimulation, n1, n2 );
   if ( class == NULL ) {
      class = CreateNewSimulationClass( res );
      HashTableAdd( p-> pTableSimulation, class ); return res;
   }
   for each node cand in class
   if ( CheckFunctionalEquivalence( cand, res ) ) {
      AddNodeToEquivalenceClass( class, res ); return cand;
   }
   AddNodeToSimulationClass( class, res ); return res;
}
```

Figure 2.8. Pseudo-code of the AIGs construction by the FRAIG method (MISHCHENKO 2005).

### 2.3.6 Comparing Methods

In this section, a simplified comparison between the presented methods for technology independent logic optimization in the logic synthesis flow is presented. Table 2.4 present a comparison of the characteristics of the presented methods.

The column “Start Point” compares the different approaches regarding the need of having either a Boolean equation in SOP form or may be any functional description. The next column, “Incompletely Specified Function” indicates if the method is able to handle Don’t Care variables. The “Multiple Solutions” column indicate if the approach may return several results for a following evaluation and the “Multiple Objective” column indicate if it is possible to consider more than one goal while performing optimizations. The last column, “Logic Sharing”, indicates if the method is able to provide logic sharing, whenever it is possible. In the case of BDDs, logic sharing may be possible depending on the specific method.

Table 2.4: Comparison between properties of the discussed methods.

| *Method* | *Start Point* | *Incompletely Specified Function* | *Multiple Solutions* | *Multiple Objective* | *Logic Sharing* |
|---|---|---|---|---|---|
| Equation Factorization | SOP | No | No | No | No |
| Using BDD | Functional | Yes | No | No | Yes |
| Equation Composition | Functional | Yes | Yes | Yes | No |
| TANT | Functional | Yes | Yes | No | Yes |
| Using AIG | SOP or Factorized Equation | No | Yes | Yes | Yes |

Although TANT networks present most of the desirable characteristics presented in Table 2.4, there are some other issues that must be considered when choosing the best approach for the technology independent step. One is the technology dependent step, i.e., the mapping algorithm that will be executed right after. There are several implemented solutions for mapping that consist in DAG / AIG covering, but are not able to properly cover a TANT if it presents large fan-in nodes. Moreover, as mentioned earlier, TANT size (number of nodes) does not correlate properly with resulting circuit area.

Moreover, none of the described methods is able to handle multiple equations and provide a logic sharing among these functions.

## 2.4 Cost Estimation

### 2.4.1 Optimization Targeting Area

Traditionally, improving circuit area is (and has always been) one key target for any well-succeeded logic synthesis tool. The reason for that is that any circuit area reduction usually reflects positively in several different optimization design goals, such as reducing circuit power consumption, reducing fabrication cost, reducing interconnections delay, improving circuit yield, among others. Moreover, circuit area is usually simpler to estimate and does not require any signal evaluation.

Therefore, all algorithms applied for circuit automatic synthesis usually present estimations of resulting circuit area. There are several ways to estimate area, and the best choice depends both on the information available and on the precision required at the evaluation moment. For instance, during the mapping stage (when the cells are chosen from a pre-characterized library) it is possible to use the precise area of the cells available and select the set of cells with the smaller area (although the effects of routing in circuit area are hard to estimate). In the technology independent step of the logical synthesis flow, where the cell area is unknown, a simpler estimation is used in order to

evaluate what is the best solution in terms of area. In this case the estimation will be performed according to the selected approach, but usually reflecting the number of nodes of a graph (in AIGs or BDDs), the number of nodes and their number of inputs (TANTs) or the number of literals and operations performed (equations).

### 2.4.2 Optimization Targeting Delay

Although targeting area for circuit optimization has presented quality results for several years, recent works are including other goals for optimization in order to either improve or favor a given characteristic of their circuit.

In a logic circuit, a path is a sequence of nets that are used to link two given points - usually linking one circuit input to one circuit output. Therefore, more than one path may exist linking a given input to a given output. Figure 2.9 shows an example of a small circuit containing two paths from input C to output Q, represented as Path 1 (dashes) and Path 2 (dots). Path 1 can be described as {C, g2, n1, g3, n2, g5, Q} while Path 2 can be described as {C, g2, n1, g4, n3, g5, Q}.

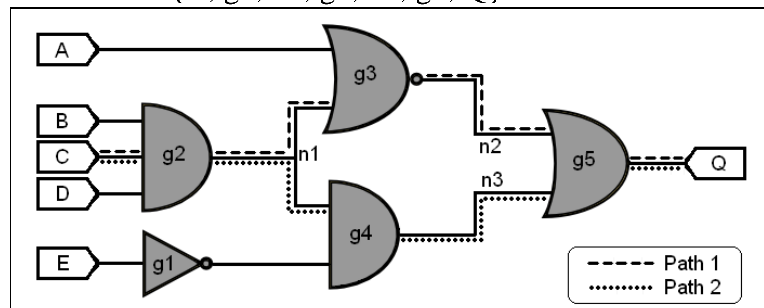



Figure 2.9. Small circuit showing two paths from input C to output Q, represented as Path 1 (dashes) and Path 2 (dots).

The effort (time) required to pass through the longest (slowest) path among all existent paths will be said to be the logical depth (complexity, cost) of this input regarding the output. Moreover, it is possible to define the logical depth of the complete circuit as the maximum of the logical depth among all inputs. There are several ways to measure logical depth, and the choice of the best measure depends both on the information available and on the precision required at the evaluation moment. For instance, during the mapping stage (when the cells are chosen from a pre-characterized library) it is possible to consider the delay of the cells, but it is hard to consider the interconnections delay. Another example is when the circuit is being globally placed and routed, where the interconnections parasites must be considered in order to obtain quality results. In the technology independent step of the logical synthesis flow, neither the cell information nor the interconnection information are available and, therefore, some simpler estimations are used in order to estimate the paths logical depth. These estimations depend on the structure used to perform the synthesis, but usually reflect the height of a graph (in AIGs or BDDs), the complexity of the elements (TANTs) or the operations performed (equations, which, in several times, are evaluated by the height of a parsing tree generated by the equation).

The logical depth is important because it is the expression of the time required to the output of the given circuit to reflect the changes in any of its inputs in a given instant. Therefore, it will affect directly the possible circuit operation frequency.

The existing algorithms for optimizing circuits are not able to handle hundreds of thousands cells at once (especially considering the amount of possible paths that this may generate and the interdependence of signals in the design. Therefore, splitting the circuit in subcircuits (as in divide-an-conquer approaches) is a common and efficient way to handle larger circuits in a feasible time. In this sense, several different strategies for circuit splitting were proposed (K-Cuts, KL-Cuts) and are not the focus of this discussion. Figure 2.10 presents a circuit long path from G to Q and a possible splitting of this circuit in three parts, (a), (b) and (c).

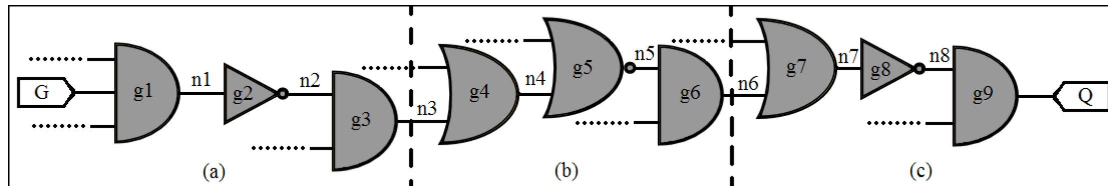


Figure 2.10. A circuit long path from G to Q and a possible splitting of this circuit in three parts, (a), (b) and (c).

The cost (delay) of the complete path may be considered as the sum of the costs attributed to each element in the path. Therefore, for any path starting in A, passing through B and ending in C, the cost function cf{{A→C} will be cf{A→B} + cf{B→C}. For instance, Figure 2.11 presents the same path than Figure 2.10, plus adding hypothetical cost to all gates (and no cost to the interconnections, as if it was in the mapping step). The (a) portion of the path will present cost cf{G→g3} = 10 + 2 + 5 = 17, the (b) portion cost cf{n3→g6} = 6 + 6 + 5 = 17 and the (c) portion cost cf{n6→Q} = 6 + 2 + 5 = 13. Therefore, the cost cf{G→Q} = cf{G→g3} + cf{n3→g6} + cf{n6→Q} = 17 + 17 + 13 = 47.

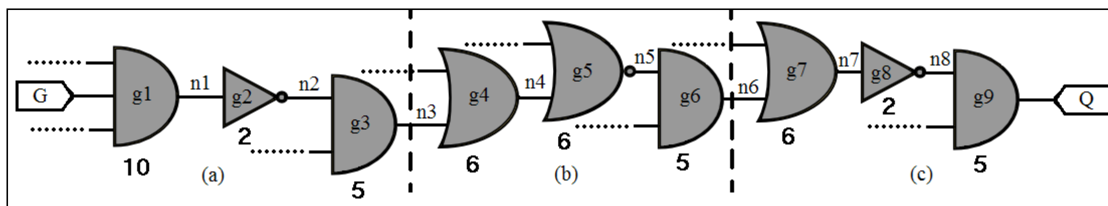


Figure 2.11. Same circuit path presented in Figure 2.10, including the hypothetical cost to the gates.

Considering this transitiveness property of the cost function, it is possible to optimize a single portion of the circuit and it will improve the circuit as a hole (as long as all paths passing the given portion are considered together). For instance, consider the circuit presented in Figure 2.9 as a subcircuit, with hypothetical costs, as shown in Figure 2.12.

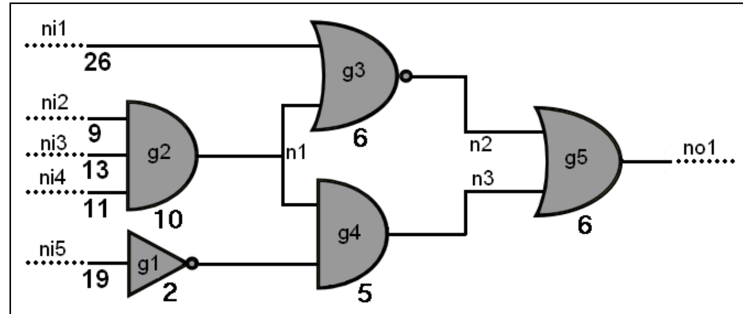


Figure 2.12. Same circuit presented in Figure 2.9 presenting costs in the inputs and gates.

The subcircuit presented in Figure 2.12 has the paths and costs presented in Table 2.5. The logical depth for each input to the subcircuit output is obtained by the maximum cost related to the input. Therefore, for *ni1* the cost is 38 (there is only one path associated) but for *ni2* it is 31 because it is the longest (more expensive) path associated to input *ni2*. The final cost of this subcircuit is the cost of the more expensive path of them all, which, in this case, is the cost of path 0, 38.

Table 2.5: The Logical Depth (Cost) of each path in the subcircuit presented in Figure 2.12.

| Path | Start | End | Elements | Logical Depth (Cost) |
|---|---|---|---|---|
| **0** | ni1 | no1 | ni1, g3, n2, g5, no1 | 26 + 6 + 6 = **38** |
| **1** | ni2 | | ni2, g2, n1, g3, n2, g5, no1 | 9 + 10 + 6 + 6 = **31** |
| **2** | | | ni2, g2, n1, g4, n3, g5, no1 | 9 + 10 + 5 + 6 = **30** |
| **3** | ni3 | | ni3, g2, n1, g3, n2, g5, no1 | 13 + 10 + 6 + 6 = **35** |
| **4** | | | ni3, g2, n1, g4, n3, g5, no1 | 13 + 10 + 5 + 6 = **34** |
| **5** | ni4 | | ni4, g2, n1, g3, n2, g5, no1 | 11 + 10 + 6 + 6 = **33** |
| **6** | | | ni4, g2, n1, g4, n3, g5, no1 | 11 + 10 + 5 + 6 = **32** |
| **7** | ni5 | | ni5, g1, n4, g4, n3, g5, no1 | 19 + 2 + 5 + 6 = **32** |

Although optimizing any other path cost might be interesting in some situations, the main goal should be optimizing path 0 since its cost (38) is the cost that will be associated to *no1* for any evaluation of the next subcircuits that uses no1 as input.

### 2.4.3 Optimizing Multiple Targets

Area optimization, for all mentioned in Section 2.4.1, continues to be an important goal in any circuit automatic synthesis process. Besides that, synthesizing a balanced circuit is desirable since the circuit delay will be its worst path delay. Moreover, other circuit characteristics such as the number of transistors in series are important and regarding more than one characteristic while synthesizing a circuit may provide much better results.

For instance, an algorithm for equation composition regarding both area (by controlling the number of literals in an equation) and number of transistors in series (by evaluating the sequence of operations in the resulting equation) is presented by Reis (REIS, 2009). Moreover, algorithms that optimize the number of nodes in a tree (either an AIG or a BDD) and try to generate a tree as balanced as possible are also available in the literature (YANG, 1999; MISCHENKO, 2006).

Hence, approaches targeting multiple objectives are rising in the last ten years. These approaches combine area and delay information either in a hierarchical improvement sequence (first improve area and among the results with smaller area obtain the one with smaller delay) or in a weighted cost function, attributing weights for each characteristic and reducing the resulting cost.

## 2.5 Related Tools

### 2.5.1 ABC Tool

ABC (BERKELEY, 2010) is a system for synthesis and verification for both combinational and sequential logic circuits appearing in hardware designs. ABC combines fast scalable logic optimization based on And-Inverter Graphs (AIG) with innovative algorithms for integrated sequential optimization and verification. ABC is meant to provide an experimental implementation of these algorithms and, at the same time, become a convenient programming environment for building similar applications in the future.

# 3 PROPOSED METHOD

The method is based on combining AIGs of simpler functions in order to obtain AIGs of more complex functions until the target function is found. Functions are represented as pairs composed by one truth table (represented as an integer) and by one AIG node that is known to represent the function. The composition of the functions is performed by applying ANDs and ORs operations in both elements of the pair. Figure 3.1 illustrates this concept. There is a subtle consequence of using the double representation. Once the operations are made in both elements of the pair, the resulting pair will have a structural representation (AIG) whose functional representation (truth table) is also known. Notice that the double operations on the pair {AIG, Truth Table} are less expensive than computing the AIG from a truth table or vice-versa. This is a core observation for the understanding of the method. The AND operation and the OR operation on AIGs are very simple, as they imply only on the creation of a single node connecting the two AIGs, as shown in Figure 3.2. Notice that the OR operation in Figure 3.2. (b) is represented as a AND node with inverting signs on all three arcs, resulting in a De Morgan equivalent of an OR-operation node.

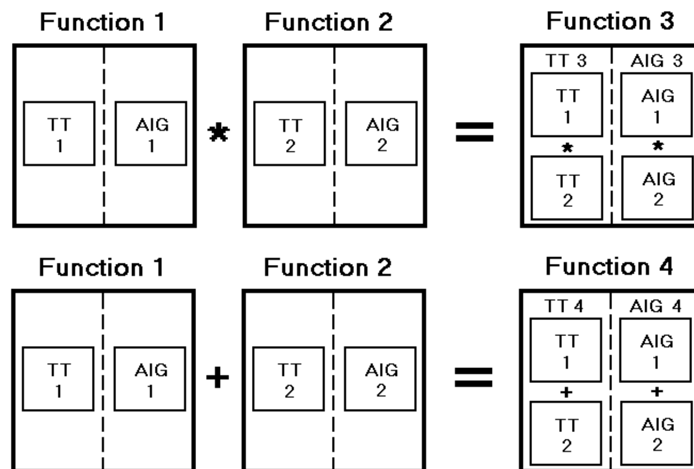



Figure 3.1. Diagram presenting how the AND and OR operations are applied to the functions during composition.

The truth table class is basically composed of the output vector of the function and the list of variables of the target function (ordered lexicographically). The AND and OR operations may be performed in a straightforward way on the output vector since the variable ordering is always the same in all functions. The AIG class consists basically in two pointers for other AIG object and one string used to store a variable

name. If the AIG object is a one variable function, it contains the variable name on the string and the two pointers are NULL. If it is not a one variable function, the string is empty and the pointers indicate the two AIGs associated to it. Since every AIG is composed only by AND nodes, the AND operation is performed by simply adding the two AIGs as children nodes of a new AIG node, which will correspond to the new graph. The OR operation is performed based on the DeMorgan property (A+B is equivalent to !(!A*!B)) and therefore the new graph may be generated by including both children AIGs inverted and inverting its output as well.

Figure 3.2 presents how two AIGs are composed by AND and OR operations.

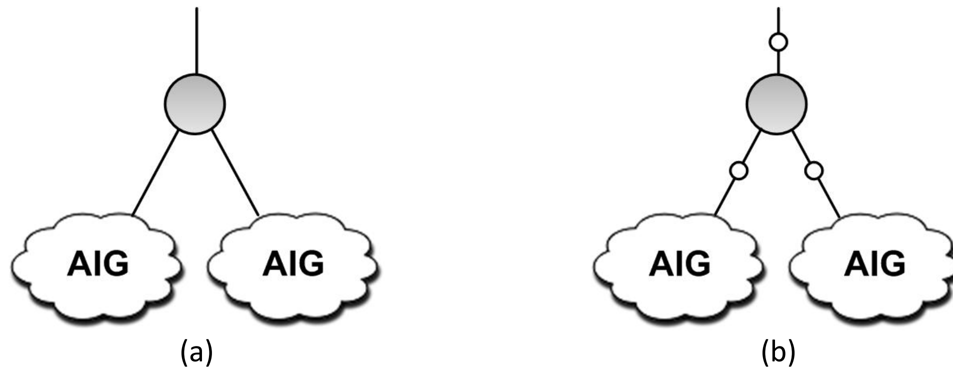



Figure 3.2. The (a) AND operation and the (b) OR operation over two AIGs, generating a new one.

The proposed algorithm is performed in three steps. The first step is to construct a set of allowed sub-functions, to reduce the search space. The second step is to create the single variable function representations that will be used as the start point of the algorithm. From the single-variable function representations, the algorithm proceeds to combine existing functions to produce new associations that will be stored if they are in the list of allowed sub-functions and discarded otherwise. These steps are described in further detail in the following sections. The pseudo-code for the proposed method is presented in Figure 3.3.

```
AIG build_aig(TruthTable target_function) {
  vector<TruthTable> allowed_functions = build_allowed_subfunctions(target_function);

  create_one_variable_subfunctions(target_function);

  while( solution not found) {
     ++bucket_number;
     combine_subfunctions(allowed_functions, bucket_number);
   }

  return solution;
}
```

Figure 3.3. The pseudo-code of the algorithm.

## 3.1 Building the Allowed Functions

In order to reduce the search space of sub-functions and, therefore, improve the performance and space required by the algorithm, a set of allowed sub-functions is

determined. The strategy to produce the allowed sub-functions is to compute the cofactors and the cube-cofactors of the target function. Moreover, it is necessary to add all the functions resulting of the AND-operation and the OR-operation between functions present in the list of cofactors and cube-cofactors. Also, the target function is also included as an allowed sub-function. Figure 3.4 presents the pseudo-code of the building allowed sub-functions algorithm.

```
vector<TruthTable> build_allowed_subfunctions(TruthTable target_function) {

    vector<TruthTable> allowed_subfunctions = build_cofactors(target_function);

    allowed_subfunctions +=  build_cube_cofactors(cofactors, target_function);

    allowed_subfunctions += combine_cofactors(allowed_subfunctions);

    return allowed_subfunctions;
}
```

Figure 3.4. The pseudo-code of the build allowed sub-functions method.

Suppose that the target function is given by Equation (3.1), and the truth table presented in Table 3.1. The cofactors are obtained by setting a single variable to a fixed logic value. For instance, making $a=1$ will result in cofactor $b+c$. The cube cofactors are obtained by setting more than one variable to a fixed logic value. For instance, making $c=0$ and $b=1$ will result in cube cofactor $a$. It is very important to notice that these operations are made directly into the truth table representation, before any AIG or equation is computed. Additionally, for the purpose of knowing if a function is allowed or not, only the functional representation (truth table) is stored. The complete set of distinct cofactors and cube-cofactors are listed in Figure 3.6.

$$F=(A+C)*(B+C) \tag{3.1}$$

First, as all truth tables may be represented by the output vector only, it is possible to represent the target function as 01010111.

Table 3.1: Truth table for the equation F = (A+C)*(B+C).

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

The calculation of the negative and the positive cofactors for the “A” variable is performed as follows: The negative cofactor is built from the values from the output vector where “A” variable is ‘0’ and these set is replicated for where the “A” variable is ‘1’. Therefore, since the values where “A” is ‘0’ are 0101, after the replication the negative cofactor of the target function for “A” is 01010101. Similarly, the positive

cofactor of the target function for the variable “A” is built based on the values of the output when “A” is ‘1’ and replicated for the values where “A” is ‘0’. Therefore, the positive cofactor for “A” is 01110111. The cofactors for the variables “B” and “C” are calculated in the same way and their values are presented in Table 3.2.

Table 3.2: Cofactors of the target function 01010111.

| Target Function | A | | B | | C | |
|---|---|---|---|---|---|---|
| | Neg. Cofactor | Pos. Cofactor | Neg. Cofactor | Pos. Cofactor | Neg. Cofactor | Pos. Cofactor |
| **01010111** | **01010101** | **01110111** | **01010101** | **01011111** | **00010001** | **11111111** |

After obtaining the cofactors, the cube-cofactors are calculated recursively. The cube-cofactor is the cofactor for a variable calculated over the values of a cofactor. For instance, the Negative Cofactor for the variable “A” may be used to generate 4 cube-cofactors (positive and negative, for variables “B” and “C”). The cube-cofactors are calculated in the same way of the cofactors. The recursion stops in two situations: a cofactor or a cube-cofactor presents a constant value (i.e. 00000000 or 11111111) or the cube-cofactor is identical to the cofactor used to calculate it. Notice that the cube-cofactor for a given variable is not calculated over the cofactor for the same variable.

Table 3.3: Cube-cofactors of the cofactors of the target function Q = A*B+C.

| | Cofactors | | Cube-cofactor in A | | Cube-cofactor in B | | Cube-cofactor in C | |
|---|---|---|---|---|---|---|---|---|
| | | | Negative | Positive | Negative | Positive | Negative | Positive |
| **A** | Neg. Cofactor | **01010101** | **-** | **-** | **01010101** | **01010101** | **00000000** | **11111111** |
| | Pos. Cofactor | **01110111** | **-** | **-** | **01010101** | **11111111** | **00110011** | **11111111** |
| **B** | Neg. Cofactor | **01010101** | **01010101** | **01010101** | **-** | **-** | **00000000** | **11111111** |
| | Pos. Cofactor | **01011111** | **01010101** | **11111111** | **-** | **-** | **00001111** | **11111111** |
| **C** | Neg. Cofactor | **00010001** | **00010001** | **00010001** | **00000000** | **01010101** | **-** | **-** |
| | Pos. Cofactor | **11111111** | **-** | **-** | **-** | **-** | **-** | **-** |

The complete vector of cofactors and cube-cofactors (which do not contain any repeated value) is presented in Figure 3.5:

*allowed_subfunctions* [**01010101, 01110111, 01011111, 00010001, 11111111, 00000000, 00110011, 00001111**]

Figure 3.5. The vector with all distinct cofactors and cube-cofactors for the target function f=A*B+C.

The final step of the process of building the allowed sub-functions is to combine the cofactors and cube-cofactors among themselves using both the AND and the OR operators (only the constant values are not considered because they add no gain in this process). Since both AND and OR operators are symmetric, it is not necessary to evaluate f1*f2 and f2*f1. Taking this into consideration reduces the effort of combining by half.

The resulting sub-functions that were not already present in the allowed sub-functions vector are then included. Table 3.4 presents the results for the AND operations for the functions presented in Figure 3.5 while Table 3.5 presents the results for the OR operations for the same set of functions. The new functions, that will be included in the set of allowed sub-functions, are underlined. The final set of allowed sub-functions is presented in Figure 3.6.

Table 3.4: AND operations applied over the cofactors and cube-cofactors resulting of previous steps.

| | 01010101 | 01110111 | 01011111 | 00010001 | 00110011 | 00001111 |
|---|---|---|---|---|---|---|
| 01010101 | - | 01010101 | 01010101 | 00010001 | 00010001 | 00000101 |
| 01110111 | - | - | 01010111 | 00010001 | 00110011 | 00000111 |
| 01011111 | - | - | - | 00010001 | 00010011 | 00001111 |
| 00010001 | - | - | - | - | 00010001 | 00000001 |
| 00110011 | - | - | - | - | - | 00000011 |
| 00001111 | - | - | - | - | - | - |

Table 3.5: OR operations applied over the cofactors and cube-cofactors resulting of previous steps.

| | 01010101 | 01110111 | 01011111 | 00010001 | 00110011 | 00001111 |
|---|---|---|---|---|---|---|
| 01010101 | - | 01110111 | 01011111 | 01010101 | 01110111 | 01011111 |
| 01110111 | - | - | 01111111 | 01110111 | 01110111 | 01111111 |
| 01011111 | - | - | - | 01011111 | 01111111 | 01011111 |
| 00010001 | - | - | - | - | 00110011 | 00011111 |
| 00110011 | - | - | - | - | - | 00111111 |
| 00001111 | - | - | - | - | - | - |

*allowed_subfunctions* [**01010101, 01110111, 01011111, 00010001, 11111111, 00000000, 00110011, 00001111, 00000101, 01010111, 00000111, 000010011, 00000001, 00000011, 01110111, 01111111, 00011111, 00111111**]

Figure 3.6. The complete vector with all allowed sub-function.

## 3.2 Create Single Variable Functions

The creation of pairs {truth table, AIG} is straightforward. These representations are necessary in the initialization phase of the algorithm, as the algorithm start with the simplest possible structures, i.e.: the descriptions of single variable {truth table, AIG} pairs. Notice that only the variable functions that are present in the allowed functions list must be created in the initialization.

Each variable has its own truth table constructed, according to their value in a truth table containing all variables of the target function, ordered in a lexicographic manner. Moreover, for each variable, its complementary value is also computed.

Considering the truth table presented in Table 3.1, the three variables “A”, “B” and “C” would present the values in Table 3.6. If is allowed to the graph to present the inverted inputs as well, the complemented values of the variables “!A”, “!B” and “!C” must also be generated.

Table 3.6: Truth tables of the variable functions and their inverted values.

| A | B | C | !A | !B | !C |
|---|---|---|---|---|---|
| **00001111** | **00110011** | **01010101** | **11110000** | **11001100** | **10101010** |

Depending on the target function, not all variables (and their inverted values) will be necessary to the composition. The selection of what functions should be kept for the composition step could be performed by evaluating the unateness of the input variables. Positive unate variables would not require their complement, while negative unate variables would not require their direct value. Binate variables would require both direct and negated values. However, the unateness evaluation may be avoided by checking the list of allowed subfunctions. Since the allowed subfunctions derive from the cofactors and cube-cofactores of the target function, by removing the functions that are not present in the allowed functions list (generated in the previous step), only the required functions are kept.. In the current example, only the elements [00001111, 00110011, 01010101] should be maintained.

## 3.3 Combining and Evaluating Functions

As it was already mentioned, the method is based on combining AIGs of simpler functions in order to obtain AIGs of more complex functions until the target function is found. The starting point is the set of known sub-functions represented by the pair {truth table, AIG} of the single variable functions. All the AIGs of those functions present zero nodes, since they are just a variable representation. Figure 3.7 illustrates the process for creating all the buckets up to 4-nodes AIGs. The first step is creating from scratch the bucket of 0-nodes AIG, which was already described in Section 3.2. Then, the 0-nodes AIGs are combined among themselves through AND and OR operations {1} to create the 1-node AIG bucket. In a similar way, the combination {2} of the elements from the 0-nodes AIG bucket with the elements from the 1-node AIG bucket creates the 2-nodes AIG bucket. The 3-nodes AIGs bucket is generated by the association of the elements from the 0-nodes bucket and 2-nodes bucket {3} as well as by operating, two by two, the elements from the 1-node bucket {4}. Finally, the 4-nodes bucket is composed of combinations among 0-nodes and 3-nodes buckets {5} and among the 1-node and 2-nodes buckets {6}. This process continues until reaching the target function.

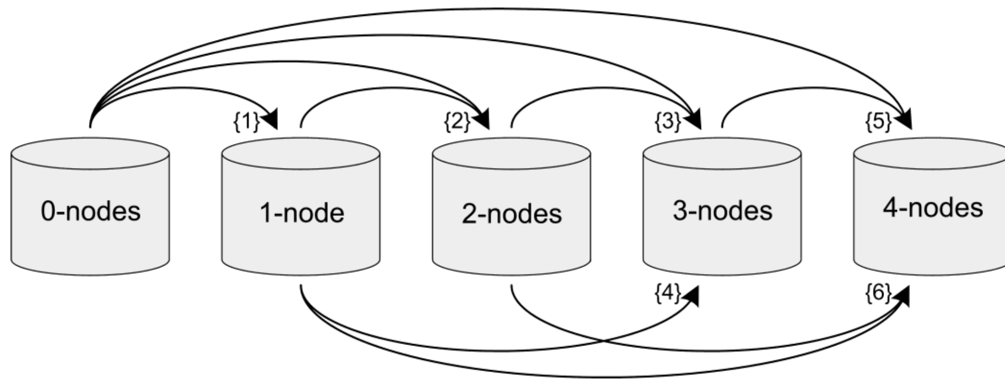


Figure 3.7. Diagram presenting how the AND and OR operations are applied to function representations during composition. The numbers in brackets indicates the algorithm step order.

The pseudo-code of the proposed algorithm for combining a given set of functions is presented in Figure 3.8. This algorithm is called several times, as shown in the while loop presented in the pseudo-code in Figure 3.1.

Since the proposed algorithm for the construction of the AIG works in a bottom up approach, the initial step consists in combining the one variable functions allowed, which are the ones presented in the first column of Table 3.7, with the other functions of one variable, presented in the second column. Whenever the combination consist in elements from the same group (same number of nodes) there is no point in combining all elements among themselves since the operations are symmetric (A+B == B+A and A*B == B*A). This last consideration reduces the operations by half.

```
map<TruthTable, AIG> combine_subfunctions(vector<map<TruthTable, AIG>> all_functions,
                                          vector<TruthTable> allowed_functions, int bucket_number)   {
  map<TruthTable, AIG> generated_functions;

  for(int i = 0; i <= bucket_number/2; ++i) {
     bucket1 = all_functions[i];
     bucket2 = all_functions[bucket_number – i];

     for each bucket1Element in bucket1
         for each bucket2Element in bucket2
             newElement = bucket1Element AND bucket2Element;

             if(newElement is allowed)
                if((newElement does not exist)
                  add newElement to generated_functions;
                else if(cost(existentElement) > cost(newElement))
                  add newElement to generated_functions;
   }
   return generated_functions;
}
```

Figure 3.8. The pseudo-code of the combining sub-functions method.

Table 3.7: OR (+) and AND(*) operations applied over the one variable functions.

| 0 Nodes | 0 Nodes | 1 Node (+) | 1 Node (*) |
|---|---|---|---|
| 00001111 | 00110011 | 00001111 + 00110011 = 00111111 | 00001111 * 00110011 = 00000011 |
| | 01010101 | 00001111 + 01010101 = 01011111 | 00001111 * 01010101 = 00000101 |
| 00110011 | 01010101 | 00110011 + 01010101 = 01110111 | 00110011 * 01010101 = 00010001 |

The generated functions that are not present in the allowed function vector generated in previous step (presented in Figure 3.4) are discarded. In this specific case, all are accepted and, therefore, are used in the next step. Table 3.8 present the resulting functions in the 0-nodes AIGs bucket and in the 1-node AIGs bucket. The next step consists in combining the elements with 0 nodes to the ones with one node, as presented in Table 3.9.

Table 3.8: OR operations applied over the cofactors and cube-cofactors resulting of previous steps.

| 0 Nodes | 1 Node |
|---|---|
| 00001111 | 00111111 |
| 00110011 | 00000011 |
| 01010101 | 01011111 |
| | 00000101 |
| | 01110111 |
| | 00010001 |

Table 3.9: OR operations applied (+) and AND(*) operations applied over the one variable functions (o nodes functions) with the 1 node functions.

| 0 Nodes | 1 Node | 2 Nodes (+) | 2 Nodes (*) |
|---|---|---|---|
| 00001111 | 00111111 | ~~00111111~~ | ~~00001111~~ |
| | 00000011 | ~~00001111~~ | ~~00000011~~ |
| | 01011111 | ~~01011111~~ | ~~00001111~~ |
| | 00000101 | ~~00001111~~ | ~~00000101~~ |
| | 01110111 | 01111111 | 00000111 |
| | 00010001 | 00011111 | 00000001 |
| 00110011 | 00111111 | ~~00111111~~ | ~~00110011~~ |
| | 00000011 | ~~00110011~~ | ~~00000011~~ |
| | 01011111 | 01111111 | ~~00010011~~ |
| | 00000101 | ~~00110111~~ | 00000001 |
| | 01110111 | ~~01110111~~ | ~~00110011~~ |
| | 00010001 | ~~00110011~~ | ~~00010001~~ |
| 01010101 | 00111111 | 01111111 | ~~00010101~~ |
| | 00000011 | 01010111 | 00000001 |
| | 01011111 | ~~01011111~~ | ~~01010101~~ |
| | 00000101 | ~~01010101~~ | 00000101 |
| | 01110111 | ~~01110111~~ | ~~01010101~~ |
| | 00010001 | ~~01010101~~ | ~~00010001~~ |

All generated functions that are not on the allowed functions set are discarded. Moreover, considering a simple cost function where the solution with the smaller number of nodes is always better, all results found for 2 nodes that were already found with 0 or 1 nodes, are discarded.

## 3.4 Cost Functions and Implications

In the third step of the presented algorithm, a cost function is used in order to evaluate which of the generated functions presented the best results for the next iterations. This cost function may consider only one aspect of the AIG, such as number of nodes or graph depth; or an association of these aspects.

In single equation factoring (REIS 2009), where sharing is not allowed, this construction can be proven optimal by using dynamic programming principles. However, the growth of AIGs does not behave this way for area (number of nodes) cost. Once the number of nodes reflects directly the final circuit area and reducing area generally improves other circuit characteristics such as power consumption and timing, presenting the smaller number of nodes is the main optimization goal of the algorithm. However, there are different solutions that may present the same number of nodes in the AIG but different graph depth. In this case, the graph with the smaller depth will lead to a circuit with smaller paths (or at least with a smaller critical path) and, therefore, better timing characteristics.

For example, consider the function represented by the truth table presented in Table 3.10. Consider that the algorithm already found a solution that corresponds to this function, represented by the AIG in Figure 3.9 (a). Instead of concluding the execution, it continues evaluating all other solutions that do not increase the number of nodes in the AIG. It will consequently find the solution presented in Figure 3.9 (b). Considering the graph depth, the AIG presented in Figure 3.9 (b) will be selected, since it presents length 2 and the other solution presents length 3.

Table 3.10: Truth table for the equation Q = A*B+C+D.

| A | B | C | D | Q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** |
| 0 | 0 | 0 | 1 | **1** |
| 0 | 0 | 1 | 0 | **1** |
| 0 | 0 | 1 | 1 | **1** |
| 0 | 1 | 0 | 0 | **0** |
| 0 | 1 | 0 | 1 | **1** |
| 0 | 1 | 1 | 0 | **1** |
| 0 | 1 | 1 | 1 | **1** |
| 1 | 0 | 0 | 0 | **0** |
| 1 | 0 | 0 | 1 | **1** |
| 1 | 0 | 1 | 0 | **1** |
| 1 | 0 | 1 | 1 | **1** |
| 1 | 1 | 0 | 0 | **1** |
| 1 | 1 | 0 | 1 | **1** |
| 1 | 1 | 1 | 0 | **1** |
| 1 | 1 | 1 | 1 | **1** |

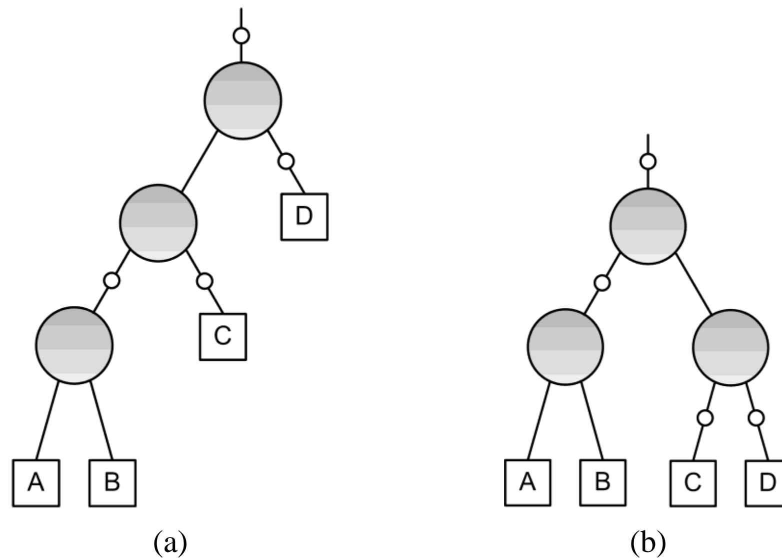



Figure 3.9. (a) An AIG resulting for the function from the truth table presented in table 3.10 and (b) another AIG for the same function, presenting the same number of nodes but a smaller graph depth.

## 3.5 Multiple Outputs

The characteristics of the AIG structure associated with the constructive approach allow the adoption of a straightforward implementation for supporting multiple outputs. This strategy consists in handling each output function as an independent target function, but maintaining the already generated structure for one function available to be shared by the next ones. The costs related to the number of nodes of the structures that are shared only are considered once, in the time of its creation. All other functions that decide to use this structure will consider the number of nodes to be zero. However, the cost associated to the logic depth must be considered in every use of the structure.

Consider, for instance, that the algorithm is processing two functions, Q1 = A*B+C and Q2 = A*B+D. The first function will be processed and generate an AIG equal to the one presented in Figure 3.11. When processing the second equation, the algorithm identifies that an intermediate graph is already available, consider its cost to be zero (in terms of number of nodes) and decides to share this structure. The resulting AIG for both outputs is the one presented in Figure 3.12.

When processing a large number of equations at a time, or really large equations, which may present logic sharing inside the own function, it is possible to generate an AIG which present one (or some) nodes with a large fun out. This characteristic may be an issue since the circuit generated using this graph would probably need a larger circuit (by either using large drive strength cells or replicating the circuit. As one of the main characteristics of the AIG is the direct relation with the resulting circuit characteristics, it may be desirable to handle this specific situation.

In this work, an algorithm for handling all nodes presenting fanout larger than a given threshold is proposed. It simple looks in the graph for every occurrence of these extremely shared nodes and replicates the subgraph starting from this node, dividing

equally the load in all copies of the structure. This algorithm is executed as a post processing and it does not represent any significant extra cost in the complete run.

## 3.6 Different Costs for Inputs

The proposed algorithm may be used to optimize entire circuits, but usually it will be used to optimize a combinational portion of a larger circuit. In this last situation, each input signal of the circuit portion may present different characteristics (e.g., different arrival times) and, therefore, building a balanced graph (such as the one discussed in Section 3.4) may not present the best results.

For this reason, the proposed method may receive, besides the target function, a list of costs for all input variables. The provided costs are used together with the costs estimated during the function composition.

For instance, consider the function presented in Table 3.11 and the cost for the inputs presented in Table 3.12.

If the algorithm do not consider the inputs cost, which means saying that all inputs present the same cost, the resulting AIG would be the one presented in Figure 3.10 (a). Notice that the graph is well balanced, presents 5 nodes and the final length of the critical path is 6 (2 nodes from the output to the variable A plus the cost 4 of the variable A). If the cost is taken into consideration, another AIG graph may present better results. Figure 3.10 (b) presents an AIG not much balanced (especially if compared with the previous one). However, the final cost for this second graph is 5 (1 node from the output to the variable A and the cost 4 of the variable A).

Table 3.11: Truth table for the equation Q = !A*B*!C*!D+!A*C*D.

| A | B | C | D | Q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** |
| 0 | 0 | 0 | 1 | **0** |
| 0 | 0 | 1 | 0 | **0** |
| 0 | 0 | 1 | 1 | **1** |
| 0 | 1 | 0 | 0 | **1** |
| 0 | 1 | 0 | 1 | **0** |
| 0 | 1 | 1 | 0 | **0** |
| 0 | 1 | 1 | 1 | **1** |
| 1 | 0 | 0 | 0 | **0** |
| 1 | 0 | 0 | 1 | **0** |
| 1 | 0 | 1 | 0 | **0** |
| 1 | 0 | 1 | 1 | **0** |
| 1 | 1 | 0 | 0 | **0** |
| 1 | 1 | 0 | 1 | **0** |
| 1 | 1 | 1 | 0 | **0** |
| 1 | 1 | 1 | 1 | **0** |

Table 3.12: Costs for the inputs in the proposed example.

| A | B | C | D |
|---|---|---|---|
| 4 | 1 | 1 | 1 |

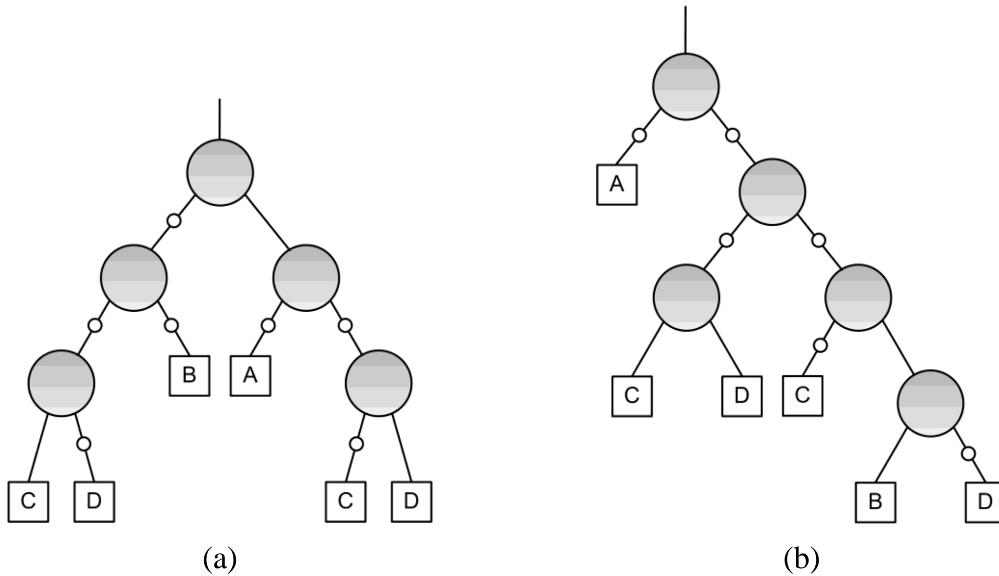



Figure 3.10. (a) An AIG resulting for the function from the truth table presented in Table 3.11 without considering the inputs costs and (b) another AIG for the same function, presenting the same number of nodes but a smaller graph depth as it considers the costs in Table 3.12.

## 3.7 Disjoint Approach

The algorithm described in previous subsections may be improved to reduce even further the search space for the solution, by using distinct buckets for functions with different orders with respect to the target function. The approach consists in separating them in three different groups of buckets, instead of comparing all generated functions among themselves: smaller than the target function, larger than the target function, and not-comparable to the target function. A function f1 is said to be larger (or smaller) than another function f2 when the on-set of f1 is a superset (or a subset) of the on-set of f2. Two functions are not-comparable when the on-sets are not contained by each other. In the disjoint effort approach, the functions in the smaller group are only associated to other functions by the OR operator, while functions in the larger group are only associated to other functions by the AND operator. The not-comparable functions are still combined by both OR and AND operators.

For instance, consider the step of building the bucket of three nodes function. In the regular approach, there are four combinations to generate the candidates:

- OR operation between one element from 0 nodes bucket and one element from 2 nodes bucket;
- AND operation between one element from 0 nodes bucket and one element from 2 nodes bucket;
- OR operation between two elements from 1 node bucket;
- AND operation between two elements from 1 node bucket.

Considering that the occupation of these buckets are the ones presented in Figure 3.11, the number of required operations to build the 3 nodes bucket would be 386.

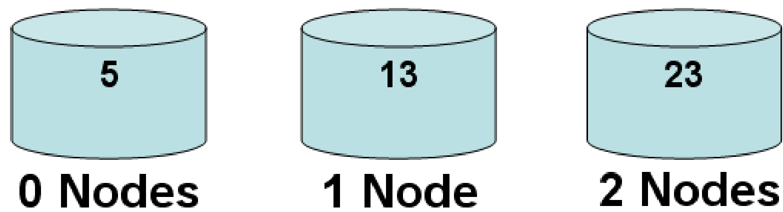



Figure 3.11. Number of elements in the buckets with 0 nodes, 1 node and 2 nodes functions in the regular approach.

In the same situation (building the bucket of three nodes function), the disjoint approach will present fourteen combinations to generate the candidates:

- OR operation between one element from smaller 0 nodes bucket and one element from smaller 2 nodes bucket;
- OR operation between one element from smaller 0 nodes bucket and one element from not-comparable 2 nodes bucket;
- OR operation between one element from not-comparable 0 nodes bucket and one element from smaller 2 nodes bucket;
- OR operation between one element from not-comparable 0 nodes bucket and one element from not-comparable 2 nodes bucket;
- AND operation between one element from not-comparable 0 nodes bucket and one element from not-comparable 2 nodes bucket;
- AND operation between one element from not-comparable 0 nodes bucket and one element from larger 2 nodes bucket;
- AND operation between one element from larger 0 nodes bucket and one element from not-comparable 2 nodes bucket;
- AND operation between one element from larger 0 nodes bucket and one element from larger 2 nodes bucket;
- OR operation between two elements from smaller 1 node bucket;
- OR operation between one element from smaller 1 node bucket and one element from not-comparable 1 node bucket;
- OR operation between two elements from not-comparable 1 node buckets;
- AND operation between two elements from not-comparable 1 node buckets;
- AND operation between one element from not-comparable 1 node bucket and one element from larger 1 node bucket;
- AND operation between two elements from larger 1 node buckets.

Although presenting a much larger set of combinations, since the number of elements in each bucket reduces, the exponential factor that controls the number of operations is significantly decreased. Considering that the occupation of the buckets are the ones presented in Figure 3.12, the number of required operations to build the 3 nodes bucket in the disjoint approach would be 238, which is a reduction of around 38% of operations when compared to the regular approach.

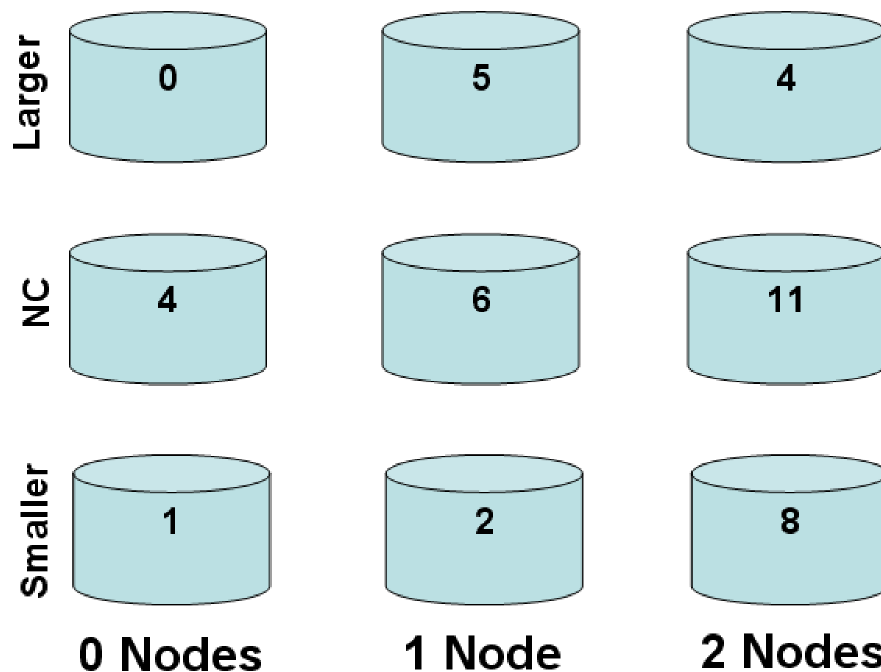



Figure 3.12. Number of elements in the buckets with 0 nodes, 1 node and 2 nodes functions in the disjoint approach.

## 3.8 Algorithm Complexity

As presented in Figure 3.3, the three main steps of the algorithm are the building of allowed subfunctions, the creation of the single variable subfunctions and the combination of subfunctions.

Although the time required to execute the building of allowed subfunctions may vary significantly on what concerns the specific characteristics of the target function, the building of allowed subfunctions complexity is quadratic to the number of literals. The cost is quadratic due to the combination using the AND and OR operations between the cofactors and cube-cofactors.

The second step of the algorithm, which is the creation of single variable subfunctions, presents a literal cost regarding the number of literals in the target function, since it is based in simple generating the functions representing the positive and negative values of the input literals.

The complexity of the third step of the algorithm, which is the combination of the subfunctions is also quadratic regarding the subfunctions available in the buckets that are combined, since it is based in performing AND and OR operations among the subfunctions available two by two.

All the improvements described in this chapter (disjoint approach and the reduction of the allowed subfunction does not affect the complexity of the algorithm, although reducing the base value of the quadratic cost. Moreover, the handling of multiple outputs and of the different weights in the inputs does not affect the complexity as well.

# 4 EXAMPLES

This chapter presents some examples of AIGs generated by the proposed method considering the different features explained in previous chapter. In some cases comparison with other approaches are also presented.

## 4.1 Main algorithm

The first example consists in a four inputs function (not read-once), represented by the equation (4.1) and by the truth table in Table 4.1.

Q=((!B*!C*D)+(B*!C*!D)+(!A*C*D)+(A*!B*!D)) (4.1)

Table 4.1: Truth table for the equation Q=((!B*!C*D)+(B*!C*!D)+(!A*C*D)+(A*!B*!D))

| A | B | C | D | Q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** |
| 0 | 0 | 0 | 1 | **1** |
| 0 | 0 | 1 | 0 | **0** |
| 0 | 0 | 1 | 1 | **1** |
| 0 | 1 | 0 | 0 | **1** |
| 0 | 1 | 0 | 1 | **0** |
| 0 | 1 | 1 | 0 | **0** |
| 0 | 1 | 1 | 1 | **1** |
| 1 | 0 | 0 | 0 | **1** |
| 1 | 0 | 0 | 1 | **1** |
| 1 | 0 | 1 | 0 | **1** |
| 1 | 0 | 1 | 1 | **0** |
| 1 | 1 | 0 | 0 | **1** |
| 1 | 1 | 0 | 1 | **0** |
| 1 | 1 | 1 | 0 | **0** |
| 1 | 1 | 1 | 1 | **0** |

The resulting graph for the ABC + FRAIG is presented in Figure 4.1 while the AIG generated by the proposed method is shown in Figure 4.2. Notice that the proposed method has presented the reduction of one node when compared to ABC + FRAIG and, besides, has reduces the graph height by one as well.

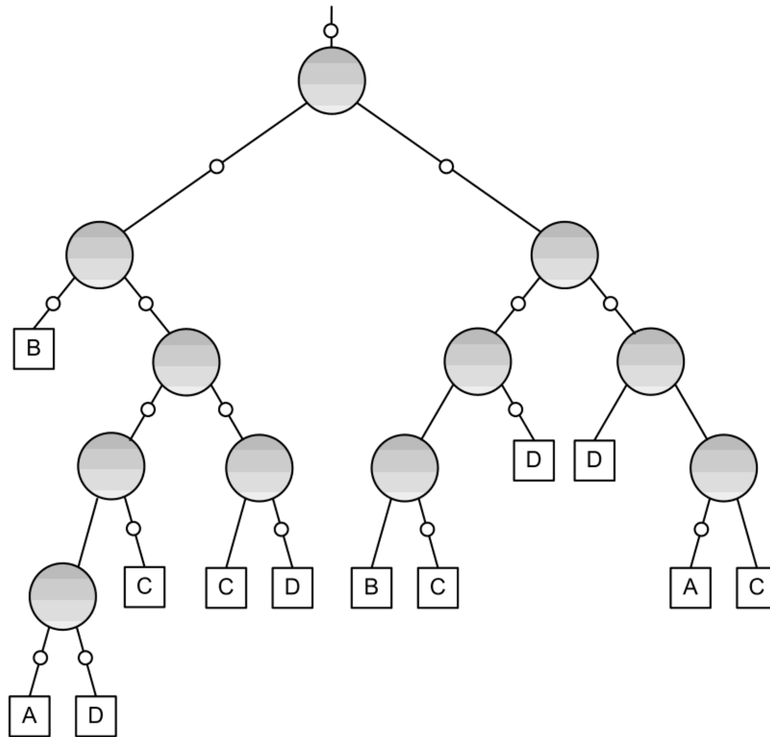



Figure 4.1. The AIG built by running ABC followed by FRAIG, for the equation (4.1).

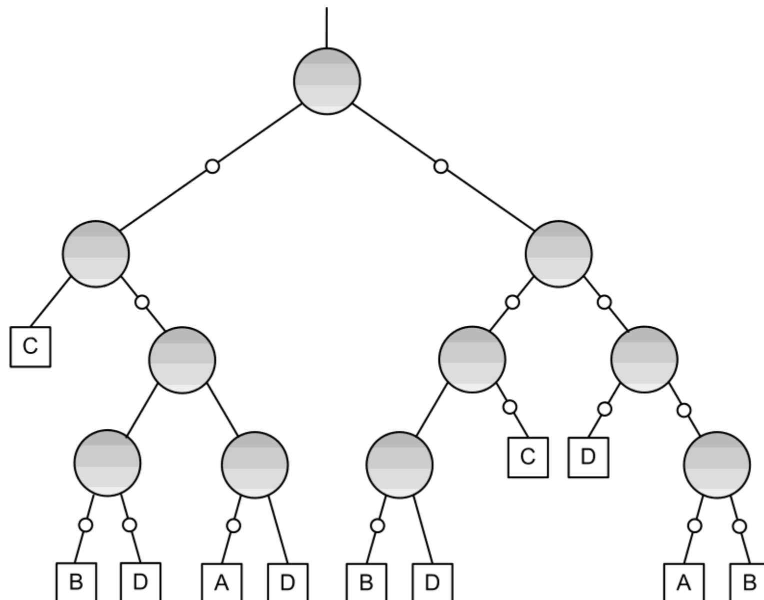



Figure 4.2. The AIG built by the proposed method for the equation (4.1).

## 4.2 Logic sharing example

The second example consists in a four inputs function (not read-once), represented by the Equation (4.2) and by the truth table in Table 4.2.

$$Q= !B*!C*D+!A*(B*C*D+!D*(B*!C+!B*C)) \tag{4.2}$$

Table 4.2: Truth table for the equation Q= !B*!C*D+!A*(B*C*D+!D*(B*!C+!B*C))

| A | B | C | D | Q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** |
| 0 | 0 | 0 | 1 | **1** |
| 0 | 0 | 1 | 0 | **1** |
| 0 | 0 | 1 | 1 | **0** |
| 0 | 1 | 0 | 0 | **1** |
| 0 | 1 | 0 | 1 | **0** |
| 0 | 1 | 1 | 0 | **0** |
| 0 | 1 | 1 | 1 | **1** |
| 1 | 0 | 0 | 0 | **0** |
| 1 | 0 | 0 | 1 | **1** |
| 1 | 0 | 1 | 0 | **0** |
| 1 | 0 | 1 | 1 | **0** |
| 1 | 1 | 0 | 0 | **0** |
| 1 | 1 | 0 | 1 | **0** |
| 1 | 1 | 1 | 0 | **0** |
| 1 | 1 | 1 | 1 | **0** |

The resulting graph for the ABC + FRAIG is presented in Figure 4.3 while the AIG generated by the proposed method is shown in Figure 4.4. Notice that the proposed method has presented the reduction of one node when compared to ABC + FRAIG and, besides, has reduces the graph height by two nodes.

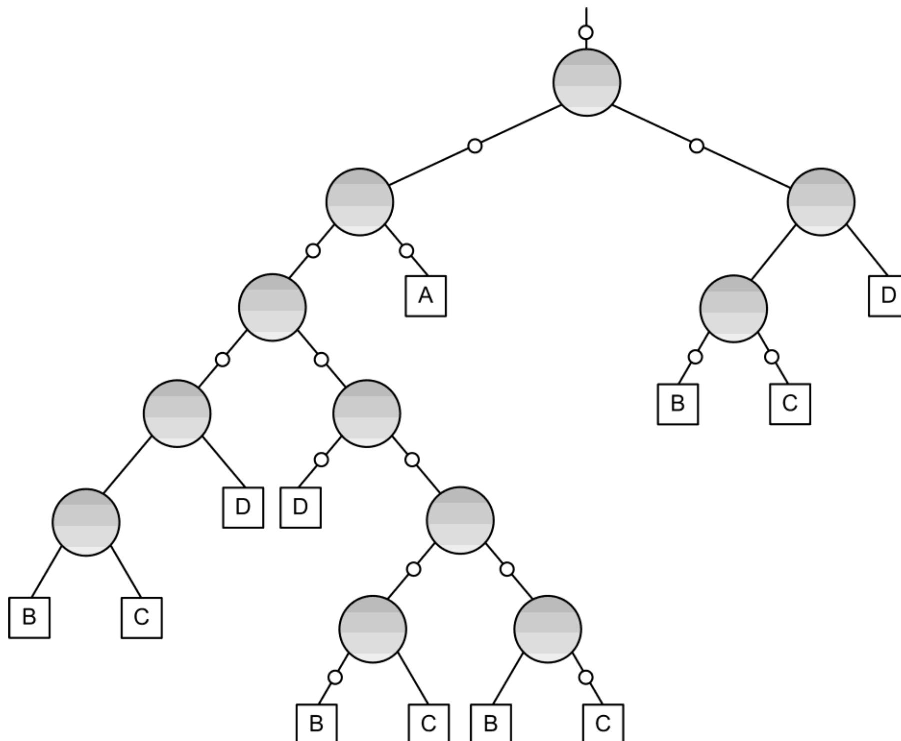



Figure 4.3. The AIG built by running ABC followed by FRAIG, for the equation (4.2).

The AIG constructed by the proposed method presents one node with fanout larger than one. This means that the proposed method was able to reuse part of the graph in more than one path, reducing the final number of nodes in the graph and, therefore, reducing resulting circuit area.

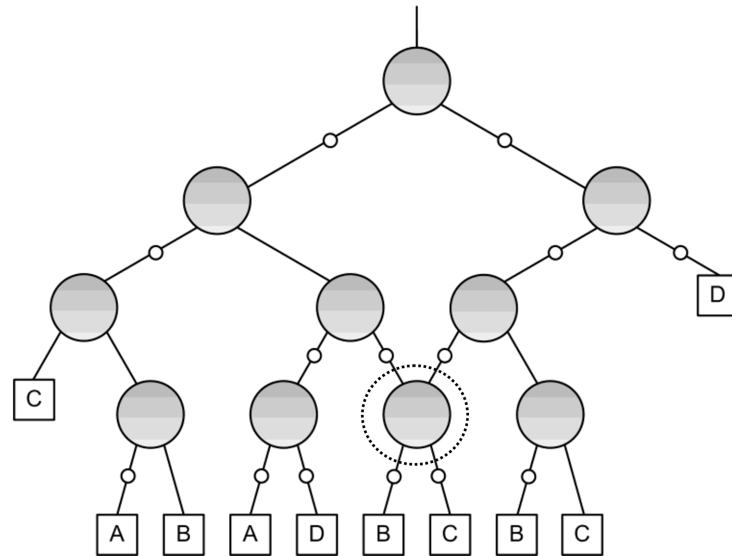



Figure 4.4. The AIG built by running the proposed method for Equation 4.2. Notice the circled node which presents fanout larger than one.

## 4.3 Multiple outputs example

The third example consists in running the proposed method for two functions at the same time (multiple outputs mode). The two functions present six inputs function (not read-once) and are represented by the Equations 4.3 and 4.4.

$$Q1=!(A+B+(C+D)*(E+F)) \tag{4.3}$$

$$Q2=!(A+B+C*D*(E+F)) \tag{4.4}$$

The resulting graphs for the execution of one equation at a time are presented in Figure 4.5 (Equation 4.3 in (a) and Equation (4.4) in (b)). Figure 4.6 presents the resulting graph for the multiple outputs mode, where both equations were processed at once. Notice that this approach reduces two nodes than the previous solutions.

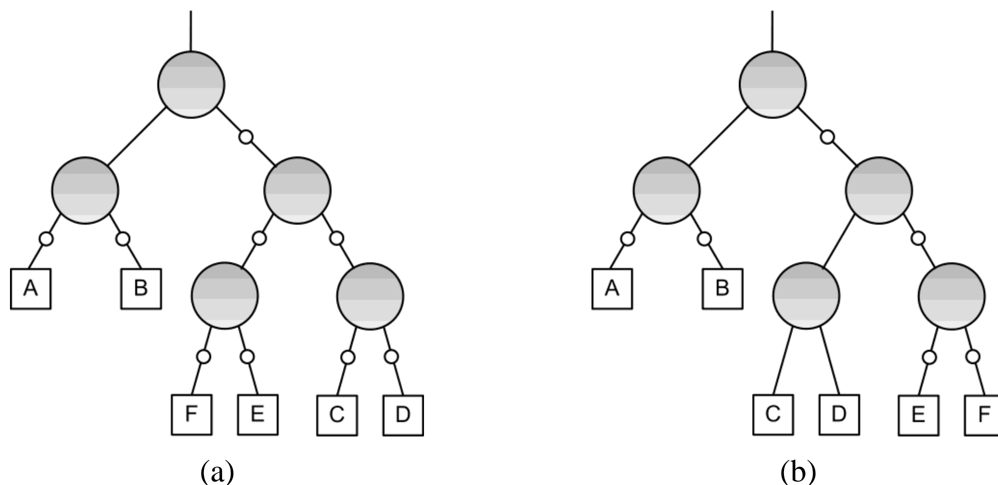



Figure 4.5. The AIGs built by running the proposed method for (a) Equation 4.3 and (b) Equation 4.4, one at a time.

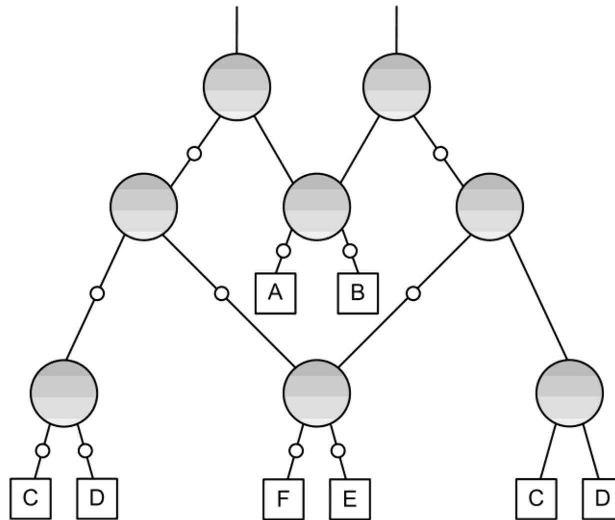


Figure 4.6. The AIGs built by running the proposed method for Equation 4.3 and Equation 4.4, both at the same time in multiple outputs mode.

## 4.4 Different input weights example

The fourth example consists in running the proposed method for a read-once 6 inputs function for 4 different input weights. The function represented by the Equation 4.5 was used as input to the algorithm for 4 different inputs costs. The costs used are shown in Table 4.3.

$$Q= !(A+B+C*D+E*F) \tag{4.5}$$

Table 4.3: Costs used as inputs costs to the AIG construction algorithm.

| Cost | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 | 1 | 3 |
| 3 | 3 | 1 | 1 | 1 | 1 | 3 |
| 4 | 3 | 1 | 1 | 4 | 1 | 3 |

The resulting graphs for the execution of each of the costs presented in Table 4.3 are presented in Figure 4.7. Figure 4.7 (a) presents the same costs for all inputs. The graph presented in Figure 4.7 (b) favors input F in order to compensate its higher cost. The AIG in Figure 4.7 (c) favors both A and F, which present higher costs in cost 3. Figure 4.7 (d) presents the result when favoring inputs A, D and F, but D with a higher cost.

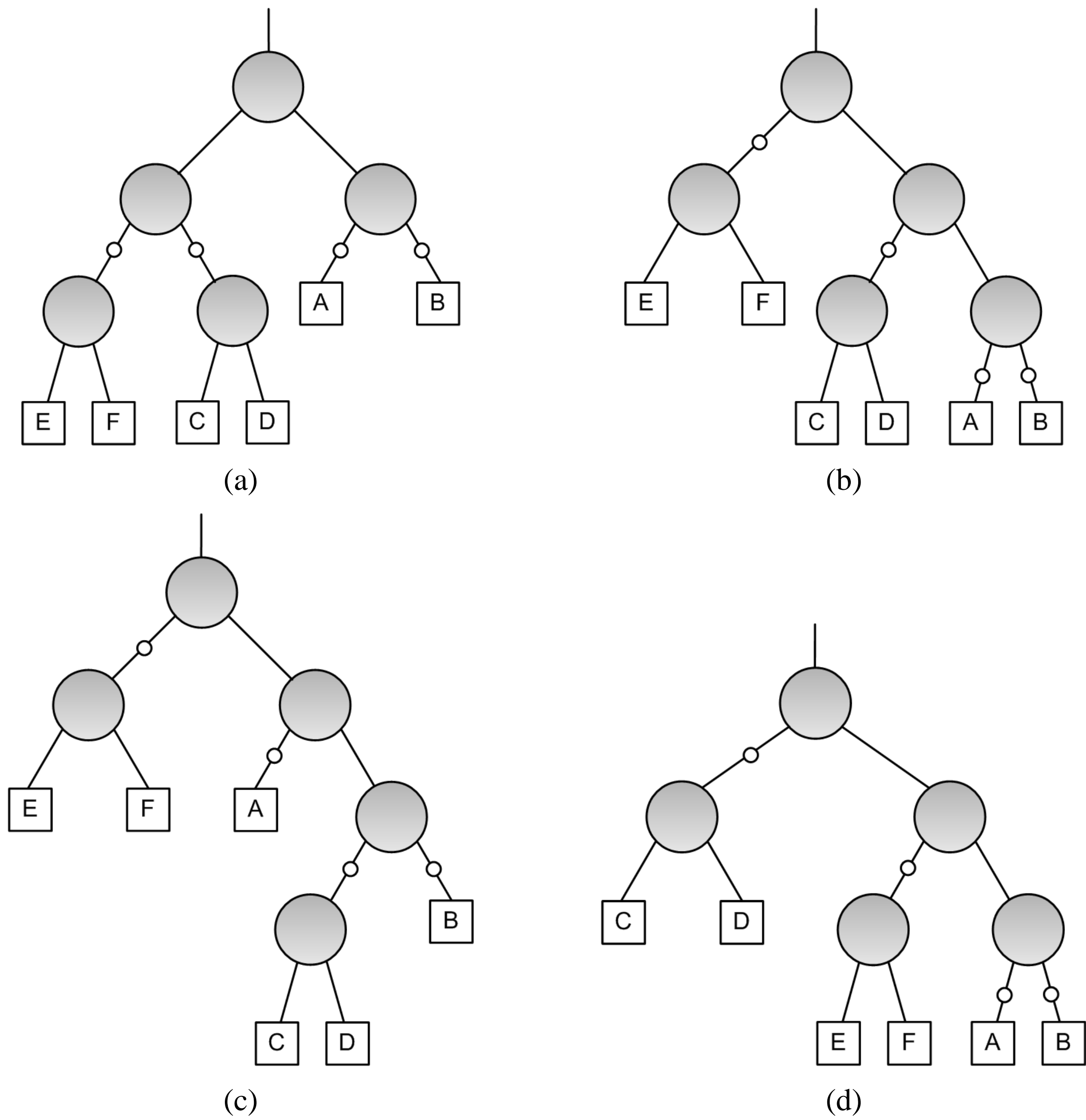



Figure 4.7. The AIGs built by running the proposed method for the function represented by Equation 4.5 and different input costs presented in Table 4.3. (a) for cost 1, (b) for cost 2, (c) for cost 3 and (d) for cost 4.

# 5 RESULTS

In order to evaluate the proposed algorithm, a comparative test was performed. Two sets of functions were created to be used in the evaluation of several aspects of the proposed method. The set A is composed of 3982 functions which present up to 4 variables while the set B is composed of 400 read-once functions presenting from 5 to 8 input variables.

## 5.1 Main Algorithm

The first test performed was regarding the main structure of the algorithm. The purpose is to evaluate if the new approach is able to produce better results than the method available in ABC for generating an optimized AIG for a single equation.

Therefore, the two sets (A and B) of functions were processed by ABC (first executing the *Good Factor "GF"* algorithm for factorization and then executing the FRAIG construction algorithm. The same functions were also processed by the Equation Composition method followed by the FRAIG construction and by the method proposed in this work.

For the set A of functions, the proposed method presented a reduction of 4.97% in total number of nodes when compared to running ABC + FRAIG and of 2.15% when compared to the Equation Composition + FRAIG. Table 5.1 presents the total of nodes generated by the three methods and the average number of nodes.

Table 5.1: Comparison of And-Inverter Graphs generated by ABC + FRAIG and the proposed method for set A of input functions.

| | **ABC + FRAIG** | **Equation Composition + FRAIG** | **Proposed Method** |
|---|---|---|---|
| **Number of Nodes** | 32813 | 31904 | 31258 |
| **Average Number of Nodes** | 8.24 | 8.01 | 7.84 |

Moreover, an evaluation of the distribution of nodes was performed between ABC + FRAIG method and the proposed method. In most cases, the two solutions presented the same number of nodes but, for over 1100 functions, the proposed method was able to reduce one node and for more than 400 functions it has reduced two nodes. for all functions and the reduction. Figure 5.1 presents the distribution of gained nodes by the proposed method when compared to ABC followed by FRAIG for the functions from set A.

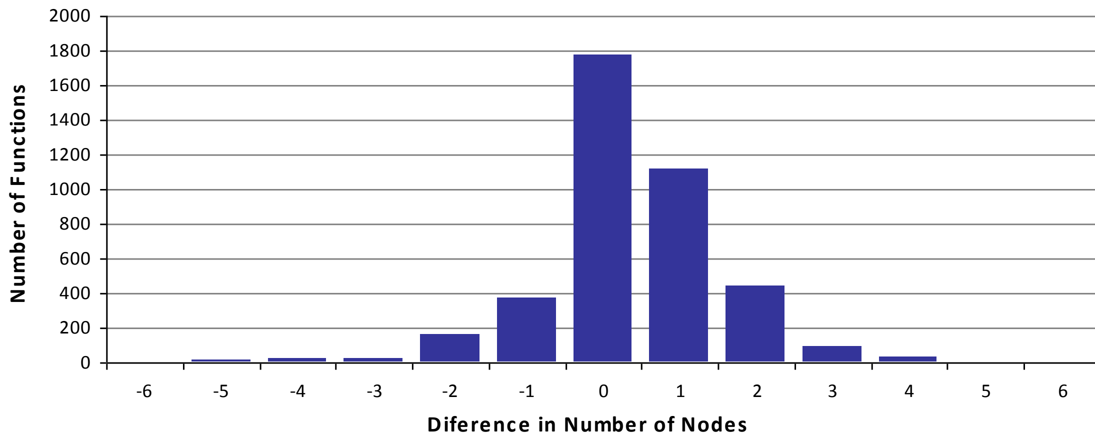



Figure 5.1. The distribution of number of nodes gained by the proposed method when compared to the ABC + FRAIG method, when applied to the input functions from set A. Negative values represent advantage to ABC + FRAIG, while positive values represent advantage to the proposed method.

For the set B of functions, the proposed method presented the same number of nodes than using Good Factor followed by FRAIG construction. This is because the set contains only read once functions, a specific type of functions that are usually easier to simplify and several methods claims to present the exactly best solution for these functions.

## 5.2 Multiple Outputs

In order to evaluate the support for multiple outputs, the set A (composed of 3982 functions) was used. The test groups these functions in sets of 2 or 4, providing to the system these sets as multiple output functions. In order to allow an easier comparison, 2 functions were removed from the total, so the amount of functions is divisible for 2 or 4. Therefore, from the 3982 functions, 3980 cells were considered for the test, creating 1990 sets of two functions and 995 sets of four functions. Table 5.2 presents the results for this comparison. A graph comparing the average number of nodes and the average logical depth is presented in Figure 5.2.

Table 5.2: Evaluation of multiple outputs result for groups of 2 or 4 functions compared to single output as well

| | **Single Output** | **2 Outputs** | **4 Outputs** |
|---|---|---|---|
| **Number of Nodes** | 31250 | 28587 | 26153 |
| **Average Number Nodes** | 7.85 | 7.18 | 6.57 |
| **Average Nodes Reduction** | - | 8.52 % | 16.31 % |
| **Sum of Logical Depths** | 15308 | 15585 | 15978 |
| **Average Logical Depth** | 3.85 | 3.92 | 4.02 |
| **Average Logical Depth Increase** | - | 1.81 % | 4.38 % |

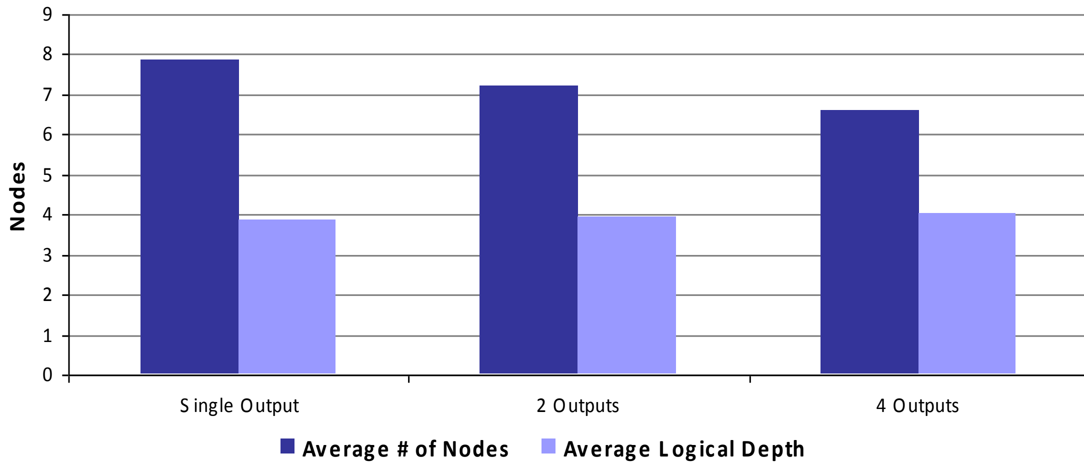



Figure 5.2. The impact of grouping functions in order to perform the multiple outputs algorithm in terms of average number of nodes and in terms of average logical depth decrease, when applied to the input functions from set A.

The results show a reduction of 8.52% in the number of nodes when grouping the functions two by two and a reduction of 16.31% when grouping the functions four by four. This reduction is a result of the sharing of nodes among the grouped functions. In this context, an increase of 1.81% in the logical depth was notice for the 2 functions sets and an increase of 4.38% in the logical depth for the 4 functions sets. This is an expected side effect of considering the number of nodes as the primary cost and the logical depth as the secondary cost and consider the nodes previous generated by other functions as of cost zero in terms of number of nodes.

## 5.3 Secondary Criterion

In order to evaluate the impact of targeting multiple objectives in the construction phase of the AIG, we proposed generating an improved equation with the Equation Composition (REIS 2009) constructive approach and apply the FRAIG algorithm for all cells present in the previous test. The logical depth is used as cost function for our approach and we compare both the number of nodes and the resulting logical depth for both approached. Table 5.3 presents the comparison for the Equation Composition followed by the FRAIG algorithm available in the ABC against the proposed method.

Table 5.3: Comparison of And-Inverter Graphs generated by Equation Composition + FRAIG and the proposed

| | **Equation Composition + FRAIG** | **Proposed Method** |
|---|---|---|
| **Sum of Logical Depths** | 17933 | 15356 |
| **Average Logical Depth** | 4.50 | 3.85 |
| **Average Logical Depth Reduction** | - | 16.88% |

A second test was performed in order to evaluate the advantages of applying secondary criterion during the AIG construction. The test described in Section 5.1 using the set B of functions presented the results shown in Table 5.4.

Table 5.4: Comparison of And-Inverter Graphs generated by Equation Composition + FRAIG and the proposed

| | **ABC + FRAIG** | **Proposed Method** |
|---|---|---|
| **Sum of Logical Depths** | 1992 | 1958 |
| **Average Logical Depth** | 4.98 | 4.89 |
| **Average Logical Depth Reduction** | - | 1.71% |

## 5.4 Different Costs for Inputs

In order to evaluate the support for different costs in the inputs, the same set composed of 3983 functions presenting up to 4 variables were used as input to the proposed method (either considering or not the inputs cost to evaluate the best AIG solution). As different costs should be applied to the input, the cost vectors presented in Table 5.5 were applied.

Table 5.5: Inputs Cost vectors used during this test.

| | **A** | **B** | **C** | **D** |
|---|---|---|---|---|
| **Vector 1** | 4 | 3 | 2 | 1 |
| **Vector 2** | 3 | 4 | 1 | 2 |
| **Vector 3** | 2 | 1 | 4 | 3 |
| **Vector 4** | 1 | 2 | 3 | 4 |

For all 3982 equations, the 4 cost vectors were used as inputs to the proposed algorithm without considering the inputs cost in the composition process of the AIG. This results in generating 15928 AIGs. The same equations and cost vectors were applied to the proposed algorithm considering the inputs cost when selecting the best AIG. Table 5.6 presents the sum of logical depths for all 15928 AIGs and the average logical depth for both cases. Notice that the algorithm that considers the inputs cost presented a reduction of 6.65% in terms of logical depth, without increasing the number of nodes.

Table 5.6: Comparison of And-Inverter Graphs generated by the proposed method either ignoring or not the inputs cost

| | **Proposed Method (Ignoring inputs cost)** | **Proposed Method (Considering inputs cost)** |
|---|---|---|
| **Sum of Logical Depths** | 121029 | 112967 |
| **Average Logical Depth** | 7,598 | 7,092 |
| **Average Logical Depth Reduction** | - | 6.65% |

## 5.5 Disjoint Approach

In order to evaluate the impact of using the disjoint approach in the proposed algorithm, another test was performed. For the same set of functions used in the previous tests, the algorithm was executed in disjoint effort mode and both the number of nodes and the number of operations were considered. Table 5.7 presents the comparison for the regular effort and disjoint effort modes of the proposed method.

Table 5.7: Comparison of And-Inverter Graphs generated by the proposed method in both regular and disjoint efforts

| | **Proposed Method (Regular approach)** | **Proposed Method (Disjoint Approach)** |
|---|---|---|
| **Number of Nodes** | 31258 | 31694 |
| **Average No. Nodes** | 7.84 | 7.95 |
| **Average Nodes Increase** | - | 1.4 % |
| **Number of Operations** | 887179 | 767042 |
| **Average No. Operations** | 2217 | 1917 |
| **Average Op. Decrease** | - | 15.65 % |

# 6 CONCLUSION

Sum of products and factored forms were used in SIS to represent logic function of single output circuit nodes. However, the granularity of the logic functions could vary, leading to optimizations in the number of literals that would not translate in better circuit characteristics after mapping. For this reason, current logic synthesis tools are using AIGs to perform the technology independent improvements in the circuit descriptions prior the mapping procedure. The AIG advantages when compared to other strategies that apply TANT networks, BDDs or equation manipulations greatly justify this tendency.

AIGs characteristics reflect closely the characteristics of the resulting circuit. There are several techniques to reduce AIGs size, available both in building time (such as FRAIG) and after construction enhancements (techniques for AIG rewriting). Moreover, AIGs are especially adequate for minimizing a cost function regarding multiple goals, such as size (number of nodes) and logic depth (graph height).

However, the techniques used to handle AIGs present some limitations, such as requiring an initial optimized (factorized or composed) equation. Therefore, some possible quality improvements are not performed, while there are other techniques that may receive functional description only, but still can not perform some possible quality improvements since they may be minimizing a cost function that do not reflect its improvement in the resulting circuit.

It is possible to join the AIG approach with the Equation Composition approach in order to implement an algorithm for performing the technology independent step in the logic synthesis flow with both logic sharing and able to handle don't care variables and independent of a previous factorization step (Figueiro 2010).

This work has proposed a novel approach for local AIG rewriting algorithm. The proposed method is based on a new synthesis paradigm (functional composition), which consists in associating simpler known pairs of functions (truth tables) and implementation (AIGs) in order to generate more complex ones.

In a first moment, a review about algorithmic logic synthesis was presented, with special interest in the technology independent step. The most frequently adopted approaches (Equation Factorization, BDDs, TANT, Equation Composition and AIGs) were presented and their positive and negative aspects were discussed. Moreover, a table comparing these methods was presented, indicating that none of the mentioned approaches were able to contemplate all aspects evaluated (start point, incompletely specified function, multiple solutions, multiple objective, logic sharing). The method proposed in this work associate two of the presented methods (functional composition +

AIGs), being able to present the qualities expressed by the previously mentioned aspects.

Since the proposed method employs as data structure a pair containing a truth table and an AIG, using the truth table as index of the functions, it does not depends on the initial description of the target function (start point). On the other hand, since the AIGs are not a canonical structure, it is possible to generate different AIGs for the same function, allowing the system to retrieve multiple solutions for the same target function, being the selection of the most appropriate solution performed by means of a cost function. The proposed algorithm combines sub-functions (both the truth table and the AIG elements of the structure) in order to reach the target function. This combination is performed applying the "AND" and the "OR" operators. During the construction, it is possible to evaluate if a given solution has already implemented the same sub-function with a smaller cost, and, therefore, eliminate the more expensive result. This bottom up approach allows the system to control the characteristics of the resulting graph by controlling the creation of the sub-graphs. Results have shown that the proposed method reduces in around 5% the number of nodes for simple functions (up to four variables), when compared to performing a factorization over an input equation and then building an AIG. For read-once functions, it was shown that the algorithm is able to produce the best solution available, as in the prior factoring algorithm proposed by (MARTINS 2010).

This work has also discussed the possible metrics for evaluating the resulting graphs and determining the cost of each solution. This cost is associated to the number of nodes of an AIG (which presents a better correlation with the resulting area of the circuit than the number of literals in an equation) and the association of graph height with paths delay. Logic depth results have shown that the algorithm is very effective in taking secondary criteria into account. The proposed method, when compared to the equation composition followed by FRAIG algorithm, presented an average decrease of 16% in the graph height (critical path length).

Another feature presented by the proposed algorithm is the possibility of considering different initial costs for the function variables, which may be used to represent the different arrival times of signals in a circuit and, therefore, that a given input (or inputs) must be favored in detriment of other(s). Results have shown that disregarding the arrival time costs may increase the average resulting logical depth in around 6.5 %.

Moreover, an approach for handling multiple output functions during the function composition is proposed, presenting good results, especially if compared to handling each input separately. A post processing algorithm for duplicating logic in case of extremely large node fanouts makes the circuits resulting from this approach feasible. The logic sharing among the functions may provide an area gain of over 8%, when processing 2 functions at once (and over 16% if processing 4 functions at once) instead of handling each function at a time.

Finally, since the proposed algorithm still presents a deficit in terms of performance, a disjoint effort approach was presented. This approach separates the sub-functions in smaller, larger or not comparable with the target function, associating the smaller ones only with the "OR" operator and the larger ones only with the "AND" operator. This leads to a significant reduction in the number of operations (around 15%), and increasing by only 1.5% the average number of nodes. We have knowledge

that this does not affect quality in single equation factoring (REIS 2009, MARTINS 2010), where sharing is not allowed. However in the current implementation based on AIGs, the final quality is being negatively affected and we have to investigate if this is intrinsic to logic sharing and cannot be solved or if it is only an implementation issue that can be fixed with a more careful ordering of the buckets taking sharing into account.

As future works, we indent to further improve the performance of the algorithm, since it is known that there are many possible enhancements in the implementation. It is also intended to explore other possible algorithmic enhancements, such as the one presented as “disjoint effort level” in this work. Moreover, the possibilities for handling multiple outputs may be further explored, especially in what concerns optimizing the target functions at the same time, trying to enhance the reuse of logic, since a sub-function may not be the best solution for none of the target functions, but, when evaluating all functions together, may present the better cost.

# REFERENCES


BERKELEY LOGIC SYNTHESIS AND VERIFICATION GROUP. **ABC: A System for Sequential Synthesis and Verification.** http://www-cad.eecs.berkeley.dey/~alanmi/abc

BJESSE, P.; BORALV, A.; **DAG-aware circuit compression for formal verification,** Proc. ICCAD'04, pp. 42-49.

BRUMMAYER, R.; BIERE, A.; **Local Two-Level And-Inverter Graph Minimization without Blowup,** In Proc. 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06), Mikulov, Czechia, October 2006.

CHATTERJEE, S.; MISHCHENKO, A.; BRAYTON, R.; **Factor cuts,** ICCAD'06, pp. 143-150.

CONG, J.; DING,Y.; **FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs.** IEEE Trans. CAD< Vol 13, No. 1 (Jan. 1994), pp. 1-12.

CONG, J.; WU, C.; DING,Y.; **Cut ranking and pruning: enabling a general and efficient FPGA mapping solution.** FPGA'99, NY, pp. 29-35.

EEN, N.; SORENSSON, N; **An extensible SAT-solver,** Proc. SAT'03, http://www.cs.chalmers.se/~een/Satzoo/

FIGUEIRO, T.; RIBAS, R.; REIS, A.; **AIG Rewriting Considering Multiple Objectives**. 25th South Symposium on Microelectronics, Porto Alegre, Brazil, May 2010.

GOLUMBIC, M.; MINTZ, A.; **Factoring Logic Functions Using Graph Partitioning.** ICCAD'99. IEE Press, Piscataway, NJ, 195-199.

GOLUMBIC, M.; MINTZ, A.; ROTICS, U.; **Factoring and recognition of read-once functions using cographs and normality.** DAC'01, ACM, New York, NY, pp. 109-114.

GOLUMBIC, M.; MINTZ, A.; ROTICS, U.; **An improvement on the complexity of factoring read-once Boolean functions.** Discrete Appl. Math, 156, 10 (May 2008), pp. 1633-1636.

HACHTEL, G. D.; SOMENZI, F., **Logic Synthesis and Verification Algorithms,** Kluver Academic Publishers (1996).

IWLS 2003; **No more counting of Literals.** Presentation of discussion Group 3 at IWLS 2003. Available at: www.sigda.org/iwls/iwls2003/no_more_literals.ppt.

JOZWIAK, L.; CHOJNACKI, A.; SLUSARCZYK, A.; **High-Quality Circuit Synthesis for Modern Technologies.** ISQED 2008, pp. 168-173.

KUEHLMANN, A.; PARUTHI, V.; KROHM, F.; GANAI, M.; **Robust Boolean reasoning for equivalence checking and functional property verification,** IEEE Trans. CAD, Vol. 21(12), 2002, pp. 1377-1394.

KUKIMOTO, E.; BRAYTON, R.; SAWKAR, P., **Delay-optimal technology mapping by DAG covering,** Proc. DAC'98, pp. 348-351.

LAWLER, E., **An Approach to Multilevel Boolean Minimization,** ACM Journal 11.3 (July 1964), pp. 283-295

LEE, H.; **An Algorithm for Minimal TANT Network Generation.** IEEE Trans. Comput. 27, 12 (Dec. 1978), pp. 1202-1206.

MARTINELLO, O.; MARQUES, F.; RIBAS, R.; REIS, A.; **KL-Cuts: A New Approach for Logic Synthesis Targeting Multiple Output Blocks.** DATE 2010, pp. 777-782.

MARTINS, M.; ROSA, L.; RASMISSEN, A.; RIBAS, R.; REIS, A.; **Boolean Factoring with Multi-Objective Goals**, in Proc. Int. Conf. Computer Design ICCD, 2010.

MISHCHENKO, A.; BRAYTON, R.; CHATTERJEE, S.; **Boolean Factoring and Decomposition of Logic Networks.** Int'l Conf. on CAD, 2008.

MISHCHENKO, A.; CHATTERJEE, S.; BRAUTON, R.; **DAG-aware AIG rewriting: a fresh look at combinational logic synthesis.** DAC' 06, 532-535.

MISHCHENKO, A.; CHATTERJEE, S.; JIANG, R.; BRAYTON, R.; **FRAIGs: A Unifying Representation for Logic Synthesis and Verification.** ERL Technical Report, EECS Dept., UC Berkeley, Mar. 2005.

MISHCHENKO, A.; SASAO, T.; **Large-scale SOP minimization using decomposition and functional properties.** DAC'03, 149-154, 2003.

PERKOWSKI, M.; CHRZANOWSKA-JESKE, M.; **Multiple-Valued-Input TANT Networks,** Proc. Multiple-Valued Logic, 1994, pp. 334-341.

PERKOWSKI, M.; CHRZANOWSKA-JESKE, M.; THUSHAR, S.; **Minimization of Multi-Output TANT Networks for Unlimited Fan-In Network Model,** Proc. ICCD'90, pp. 360 - 363, 1990.

REIS, A.; RASMUSSEN, A.; ROSA, L.; RIBAS, R.; **Fast Boolean Factoring with Multi-Objective Goals.** International Workshop on Logic & Synthesis, IWLS 2009.

SASAO, T.; **On the Complexity of Three-Level Logic Circuits**, Proc. Intern. Workshop on Logic Synthesis, MCNC, ACM SIGDA, 1982, paper 10.2.

SENTOVICH, E.; SINGH, K.; LAVAGNO, L.; MOON, C.; MURGAI, R.; SALDANHA, A.; SAVOJ, H.; STEOHAN, P.; BRAYTON, R.; SANGIOVANNI-VINCENTELLI, A.; **SIS: A system for sequential circuit synthesis.** Tech. Rep. UCV/ERL M92/41. UC Berkeley, Berkeley, 1992.

VEMURI, N.; KALLA, P.; TESSIER, R.; **BDD-based Logic Synthesis for LUT-based FPGAs**, ACM Trans. On Design Autormation of Electronic Systems, Vol. , No.4, October 2002, pp. 501-525.

VINK, H.; **Minimal TANT Networks of Functions with Don't Cares and Some Complemented Input Variables,** IEEE Trans.Comp., Vol. C-27, No. ll., Nov. 1978.

YANG, C.; SINGHAL, V.; CIESIELSKI, M.; **BDD Decomposition for Efficient Logic Synthesis**, ICCD 1999, pp. 626-631.

# APPENDIX SÍNTESE LÓGICA INDEPENDENTE DE TECNOLOGIA VISANDO MÚLTIPLOS OBJETIVOS, APLICADA A FUNÇÕES DE MÚLTIPLAS SAÍDAS, EMPREGANDO COMPOSIÇÃO FUNCIONAL DE AIGS

O emprego de ferramentas de automação de projetos de circuitos integrados permite cada vez mais que projetos complexos atingissem *time-to-market* e custos de produção factíveis. Uma das abordagens de projeto mais freqüentemente adotadas no desenvolvimento de circuitos integrados de aplicação especifica (ASIC) é o projeto baseado em biblioteca de células, também conhecido como *standard cell*. Este fluxo de projeto consiste em compor o circuito pela associação de instancias de células presentes em um conjunto pré-definido. Esta abordagem leva a uma redução do tempo de projeto uma vez que as células são construídas uma só vez e são replicadas muitas vezes em um mesmo projeto e mesmo em diferentes projetos.

Um fluxo de projeto baseado em biblioteca de células consiste em uma sequência de passos que partem de uma descrição abstrata do circuito e culminam no circuito completo, apresentando todos os seus elementos bem especificados, posicionados e roteados. Este fluxo pode ser dividido em dois blocos: síntese lógica e síntese física. A síntese lógica é responsável pela otimização da descrição original do circuito e pela seleção das células mais apropriadas para descrevê-lo. A síntese física é responsável pelo posicionamento das células e o roteamento de suas interconexões no circuito.

A síntese lógica é um processo para geração de uma implementação de um *design* a partir de uma descrição abstrata (tipicamente uma descrição estrutural ou em RTL). Esta descrição abstrada deve prover as informações a respeito do comportamento lógico esperado do circuito a ser sintetizado.

Em geral, o algoritmo de síntese lógica se divide em duas etapas, uma delas executada sobre uma descrição lógica do circuito (independente de qualquer propriedade física) e outra em que a lógica resultante da etapa anterior é mapeada na biblioteca de células ou em outra implementação física. A primeira etapa é conhecida como independente de tecnologia e a segunda como dependente de tecnologia.

Esta primeira etapa, dita independente de tecnologia, pode ser tanto uma abordagem em dois níveis quanto uma abordagem multi-nivel. A abordagem em dois níveis consiste em representar a função como uma SOP ou POS, o que significa dizer que o primeiro nível contém os termos produto e o segundo nível os termos soma (no caso da POS, o primeiro nível contem os termos soma e o segundo nível os termos

produto). Algumas inversões podem ser necessárias nas entradas dos termos do primeiro nível.

Por sua vez, a abordagem multi-nível, em geral, é composta de diversar operações lógicas, tais como decomposição, extração, fatoração, substituiçao e eliminação (HACHTEL, 1996). Estas operações podem ser tanto explícitas (SENTOVICH, 1992) ou implicitas, como na ferramenta ABC (BERKELEY 2010). O resultado do passo independente de tecnologia é uma descrição abstrata melhorada.

A síntese independente de tecnologia consiste em gerar uma representação de um dado circuito com um custo otimizado, estimado independentemente de aspectos da tecnologia. Este custo pode ser relacionado com o número de literais em uma equação ou com o número de nodos de uma TANT (LEE 1978, PERKOWSKI 1990), de um BDD (YANG, 1999 – VEMURI, 2002) ou de um AIG (KUEHLMANN, 2002, MISHCHENKO 2005, FIGUEIRO 2010). Na ferramenta SIS (SENTOVICH 1992), o custo do passo independente de tecnologia é baseado no número de literais de uma equação. Em ferramentas mais recentes, como o ABC (BERKELEY 2010), o custo é baseado em número de nodos de um AIG.

A segunda etapa do fluxo de síntese lógica, dita dependente de tecnologia, utiliza a descrição aprimorada oriunda da etapa anterior a fim de construir um DAG (se este já não lhe é fornecido) e realizar o mapeamento tecnológico sobre esse grafo. O resultado desta etapa (e consequentemente, da síntese lógica como um todo) é um circuito descrito como uma rede de portas lógicas selecionadas dentre as disponíveis em uma biblioteca de células e a informação de como essas portas devem ser interligadas.

Durante muitos anos, o passo independente de tecnologia foi executado através de algoritmos de fatoração de equações. Diversas heuristicas foram propostas para a fatoração e algumas delas atingiram alto sucesso comercial. Dentre estas se inclui o algoritmo *quick_factor* (QF) e o *good_factor* (GF), ambos disponíveis na ferramenta SIS. A maioria dos algoritmos de fatoração propostos para o passo independente de tecnologia da síntese lógica recebem como entrada uma soma de produtos (SOP). Neste caso, o processo de fatoração não é completamente otimizado (não é exato, pois depende de como os *don´t cares* foram tratados na criação da SOP).

Recentemente, métodos de fatoração que produzem resultados exatos para funções que possam ser representadas sem repetição de literais (funções *read-once*) foram propostos (GOLUMBIC, 2001) e melhorados (GOLUMBIC 2008). Contudo, o algoritmo proposto nestes trabalhos (IROF) é restrito a funções *read-once*, o que limita sua aplicação em projetos reais mais complexos.

Uma nova abordagem baseada na composição de funções foi apresentada em (REIS, 2009). O algoritmo apresentado opera por combinação de equações de uma maneira *bottom-up*, até gerar a equação que representa a função alvo. Outra abordagem é o emprego de diagramas de decisão binária (BDDs) para realizar as otimizações independentes de tecnologia. A vantagem desse método está principalmente na existencia de métodos bem conhecidos para lidar com BDDs e o fato de BDDs não dependerem de como a função de entrada foi descrita. Contudo, o processo de mapeamento de circuitos sobre BDDs é muito mais complexo do que sobre outras estruturas de grafos acíclios dirigidos (DAGs).

*Three-level And-Not Networks* (TANTs) também foram consideradas como uma alternativa para a fatoração a fim de se realizar otimizações independentes de

tecnologia. As principais vantagens das TANTs são: a possíbilidade de se obter compartilhamento de lógica e; sua estrutura independe da descrição de entrada da função. Entretando, TANTs não são apropriadas para a maioria das implementações que visam otimizar mais de um objetivo (como, por exemplo, buscar melhorar a área do circuito juntamente com o atraso de propagação, etc.) e as características de um grafo TANT não se refletem tão diretamente no circuito mapeado, em especial pelo grande *fan-in* que alguns de seus nodos constumam apresentar.

Os algoritmos de fatoração apresentados em (GOLUMBIC, 1999; REIS, 2009) não incluem reuso de sub-expressões. O reuso de sub-expressões faz com que representação multi-nível, como TANT (LEE, 1978), sejam mais competitivas que expressões de dois níveis (MISCHENKO, 2003). As propriedades de reuso de sub-expressões presentes nas representações multi-nível levam a um outro emprego interessante destes métodos, que é a síntese que funções com multiplas saídas. O suporte a multiplas saídas viabiliza o compartilhamento de lógica entre as saídas (no lugar de tratá-las como funções independentes), melhorando aspectos temporais e de consumo através da redução da área do circuito.

O desenvolvimento de novos algoritmos multi-nível pendeu recentemente para o emprego de AIGs (MISHCHENKO, 2006). AIGs são grafos acíclicos dirigidos que são compostos exclusivamente de ANDs de duas entradas e inversores. Os inversores costumam ser representados como um indicador especial nos arcos do grafo, e, portanto, todos os nodos do grafo representam portas AND. AIGs são outra maneira multi-nível de representar uma função Booleana. Seu tempo de construção e seu tamanho são proporcionais ao tamanho do circuito final. As características dos AIGs estão fortemente ligadas as características do circuito final. Por estas razões os algoritmos mais recentes de síntese lógica empregam AIGs no seu funcionamento.

Uma abordagem para a obtenção de AIGs otimizados é a construção de um AIG inicial que represente a função alvo e seu posterior aprimoramento através de técnicas de reescrita (AIG rewriting) (MISHCHENKO 2006, BRUMMAYER 2006). Uma técnica de reescrita corrente é a implementação de um algoritmo guloso que reduz o tamanho do AIG através de buscas iterativas por subgrafos que são substituidos por equivalentes reduzidos pré-computados. A técnica de reescrita apresentada por (MISHCHENKO 2006) extende o trabalho de (BJESSE 2004), ao restringir a reescrita para preservar o número de níveis lógicos e ao balancear os AIGs pelo emprego de técnicas algébricas de redução de altura de árvores.

Outra abordagem disponível para melhorar AIGs é conhecida como *Functional Reduced And-Inverter Graph* (FRAIG) (MISHCHENKO 2005). Ela está disponível na ferramenta ABC e pode ser usada para gerar o primeiro AIG antes dos procesimentos de reescrita. A técnica FRAIG consiste em um método para a contrução de AIGs a partir de uma forma fatorada de uma equação Booleana. O algoritmo emprega uma tabela *hash* estrutural para garantir que nodos equivalentes não sejam replicados. Desta forma, pode realizar melhorias em um nível (considerando permutação das entradas) ou em dois níveis (considerando permutação de nodos) (KUEHLMANN, 2002). O segundo passo realiza a detecção de equivalências funcionais por meio de um avaliador de satisfabilidade (*SAT solver*). O AIG resultante é consideravelmente reduzido (se comparado com o procedimento convencional) sem penalizar a performance, uma vez que as otimização são realizadas em tempo de construção. Além disso, FRAIGs dão ao

AIG uma forma semi-canonica, uma vez que diversas equivalências são detectadas e traduzidas para o mesmo AIG.

Nos últimos anos, portanto, foi demonstrado que AIGs são uma estrutura conveniente para emprego em algoritmos de síntese lógica. Isto se deve a alta correlação de suas características com as do circuito final (número de nodos se aproxima da área e altura do grafo se aproxima da complexidade do caminho crítico) (MISHCHENKO, 2006). Contudo, ainda existe espaço para melhorias em termos de área com algoritmos baseados em AIGs (JOSWIAK, 2008). Neste contexto, existe a necessidade de um algoritmo Booleano para reescrita de AIGs.

O objetivo deste trabalho é apresentar um método para realizar a síntese lógica independente de tecnologia pelo emprego de AIGs, os associando ao novo paradigma de síntese lógica conhecido como *functional composition* (REIS, 2009). O método é baseado na combinação de AIGs de funções mais simples a fim de obter AIGs de funções mais complexas até que a função alvo seja gerada. As funções são representadas como pares compostos de uma tabela verdade (representada como inteiros) e um nodo AIG que é a raiz da árvore que representa a função associada. A composição de funções é feita através das operações AND e OR, tanto entre as tabelas verdade quanto entre os grafos AIG. Essas duas operações são rápidas pois a associação entre AIGs se trata apenas de acrescentar um nodo associando os dois grafos anteriores e a associação entre tabelas verdade são operações lógicas entre os inteiros que representam seus vetores de saída.

A tabela verdade é representada basicamente pelo seu vetor de saída e pela lista de variáveis na ordem em que aparecem na tabela (neste trabalho todas as tabelas verdades tem suas entradas ordenadas lexicograficamente). O AIG consiste basicamente de dois ponteiros para outros nodos AIGs e uma *string* para guardar o nome da variável associada (para os nodos folha). Os arcos contém a indicação se são inversores ou não. A operação OR é realizada con o mesmo nodo AND (não existe outro disponível em um AIG) a partir de uma equivalencia por DeMorgan (A+B é equivalente a !(!A*!B)).

O algoritmo proposto é composto de três passos. O primeiro consiste em construir um conjunto de funções permitidas, a fim de reduzir o espaço de soluções. O segundo consiste em criar as funções de uma única variável, que serão usadas como as sementes para a associação *bottom-up* das funções. Por fim, o terceiro passo consiste em combinar as funções iniciais, pela aplicação dos operadores AND e OR, até que a função alvo seja encontrada. Neste passo, só são consideradas as funções intermediárias geradas que estejam presentes na lista de funções permitidas.

No terceiro passo do algoritmo apresentado, uma função custo é usada a fim de avaliar qual das soluções obtidas apresenta o melhor resultado. A solução escolhida será armazenada para as próximas iterações. A função custo pode considerar diversos aspectos do AIG. Neste trabalho foi empregado o número de nodos como critério principal e a profundidade do grafo como critério secundário.

Uma técnica para suporte a múltiplas saídas (funções de multiplas saídas) também é apresentada neste trabalho. A estratégia consiste em processar uma saída de cada vez e, a cada nova saída, considerar disponíveis para reuso as estruturas geradas pelas anteriores. Assim, o custo relativo ao número de nodos é computado apenas uma vez, sendo visto como zero para os reusos pelas demais funções (o custo em termos de

profundidade do grafo é preservado, pois este será associado de maneira independente ao atraso de cada saída).

Devido a complexidade dos projetos de circuitos integrados, dificilmente um algoritmo de síntese é utilizado para otimizar um circuito completo. Assim, em geral, a otimização é realizada por blocos, e a união desses blocos gera a síntese completa do circuito. Como diferentes sinais, oriundos de diferentes blocos predecessores, podem apresentar características muito distintas, é interessante que um algoritmo de síntese seja capaz de lidar com estimativas de diferentes atrasos dos sinais. Por exemplo, neste trabalho, um AIG balanceado é apropriado para quando todos os sinais de entrada apresentam tempos de chegada próximos. Contudo, no caso de algum sinal apresentar um atraso muito maior que os demais, esta entrada deve ser favorecida, a fim de que o novo bloco apresente um melhor atraso. Portanto, o método proposto é capaz de receber, além da função alvo, uma lista de pesos para os sinais de entrada e é capaz de considerar essa informação em tempo de construção do AIG final.

A abordagem consiste, portanto, em construir o AIG que representa a função alvo a partir da associação de AIGs mais simples, em uma estratégia *bottom-up*. O método controla as características dos grafos construídos (tanto o final quanto os intermediários) através de uma função custo que avalia a qualidade dos AIGs (esta função emprega o número de nodos como critério principal e a altura do grafo como critério segundário de avaliação). Este trabalho também discute a possibilidade de considerar um custo inicial para as variáveis de entrada da função, o que pode ser usado para representar diferentes tempos de chegada desses sinais (oriundos de outros circuitos) e, assim, utilizar essa informação para favorecer temporalmente algumas entradas em detrimento de outras. Além disso, uma abordagem para tratar funções com multiplas saídas é apresentada, propiciando o compartilhamento de lógica entre as saídas e, assim, reduzindo o número de nodos necessários para descrevê-la. Um algoritmo de pós-processamento para duplicação de lógica em caso de nodos com fan-out muito grandes é empregado para preservar a correlação das características do grafo com as do circuito resultante.

Uma vez que o método proposto emprega como estrutura de dados um par contendo uma tabela verdade e um AIG, ele é independente da descrição da função alvo. Por outro lado, uma vez que os AIGs não são canônicos, é possível gerar diferentes soluções para posterior avaliação através de uma função custo. Esta avaliação é feita em tempo de construção, não sendo honerosa ao algoritmo uma vez que apenas a sub-função com melhor custo é mantida para as etapas posteriores. Os resultados mostram uma redução de 5% no número de nodos de uma única função se comparado ao resultado da ferramenta ABC (fatoração da equação seguida da construção de um FRAIG).

Este trabalho discute as métricas possiveis para avaliar os grafos resultantes e determiner o custo de cada solução. O custo é avaliado pelo número de nodos do AIG e sua associação com a área resultante do circuito e pela altura do grafo e sua associação com o atraso dos caminhos críticos. As alturas dos grafos resultantes demonstram que o algoritmo é bem sucedido ao levar este segundo critério em consideração. O método proposto, quando comparado com algoritmos de composição de equações (REIS, 2009) seguido da construção de FRAIG, apresentou uma redução méria de 16% na altura média dos grafos, sem perda em número de nodos. Ao se permitir levar em consideração custos distintos nas entradas, foi possível reduzir a profundidade lógica média dos circuitos gerados em aproximadamente 6,5%. Os resultados obtidos com a

abordagem proposta para multiplas saídas são promissores. O compartilhamento de lógica ao se tratar duas funções ao mesmo tempo apresentou um ganho de área médio de 8% e, ao se tratar quatro funções, de 16%. Finalmente, uma abordagem disjunta foi apresentada a fim de melhorar a performance do algoritmo. Esta abordagem serapa as funções menores, maiores e não comparáveis com a função alvo. As funções menores são associadas somente pelo operador “OR” e as maiores somente pelo operador “AND”. Esta abordagem reduz em média 15% o número de operações necessárias, aumentando em menos de 1,5% o número de nodos no grafo final.

# ANNEX DESCRIPTION OF THE CLASSES AND CLASS DIAGRAM OF THE PROPOSED METHOD

The proposed method was implemented in C++, using an object oriented programming paradigm. The class diagram presenting the implemented classes, their relation and the most relevant methods is presented in Figure 1. In this description, both the class diagram and the classes description omits the sets and gets methods for simplicity.

## Function_Composer class

This is the main class of the system, which must be instantiated in any program that wish to use the method described in this work. All the configuration concerning the way the algorithm works is selected in this class.

**Attributes**:

*map<string, int>* **allowed_subfunctions**: A list of all sub-functions that may be generated for implementing a given function. This list was implemented as a map in order to improve the searching time, since it is just built once and consulted several times.

*map<string, double>* **already_generated**: Indicates all solutions that were already generated in any bucket already processed by the function_composer object, using as key the truth table converted to a string and as element the current cost, in order to use a fast comparison.

*map<string, double>* **variable_weight**: Stores all input variable weights. This information is used through the algorithm in order to determine the cost of every evaluatd solution.

*int* **effort_level**: Indicates the effort level code for running the algorithm.

**Methods**:

*bucket_element* **compose_function**(*string equation*): performs the composition of the function. It is the main method of this application, calling all the internal methods that calculate the allowed functions and coordinating the buckets construction.

*vector<bucket_element>* **compose_multiple_functions** (*vector<string> equation*): Method implemented to handle multiple functions at a time. It uses the compose function and handles the data of logic sharing between functions.

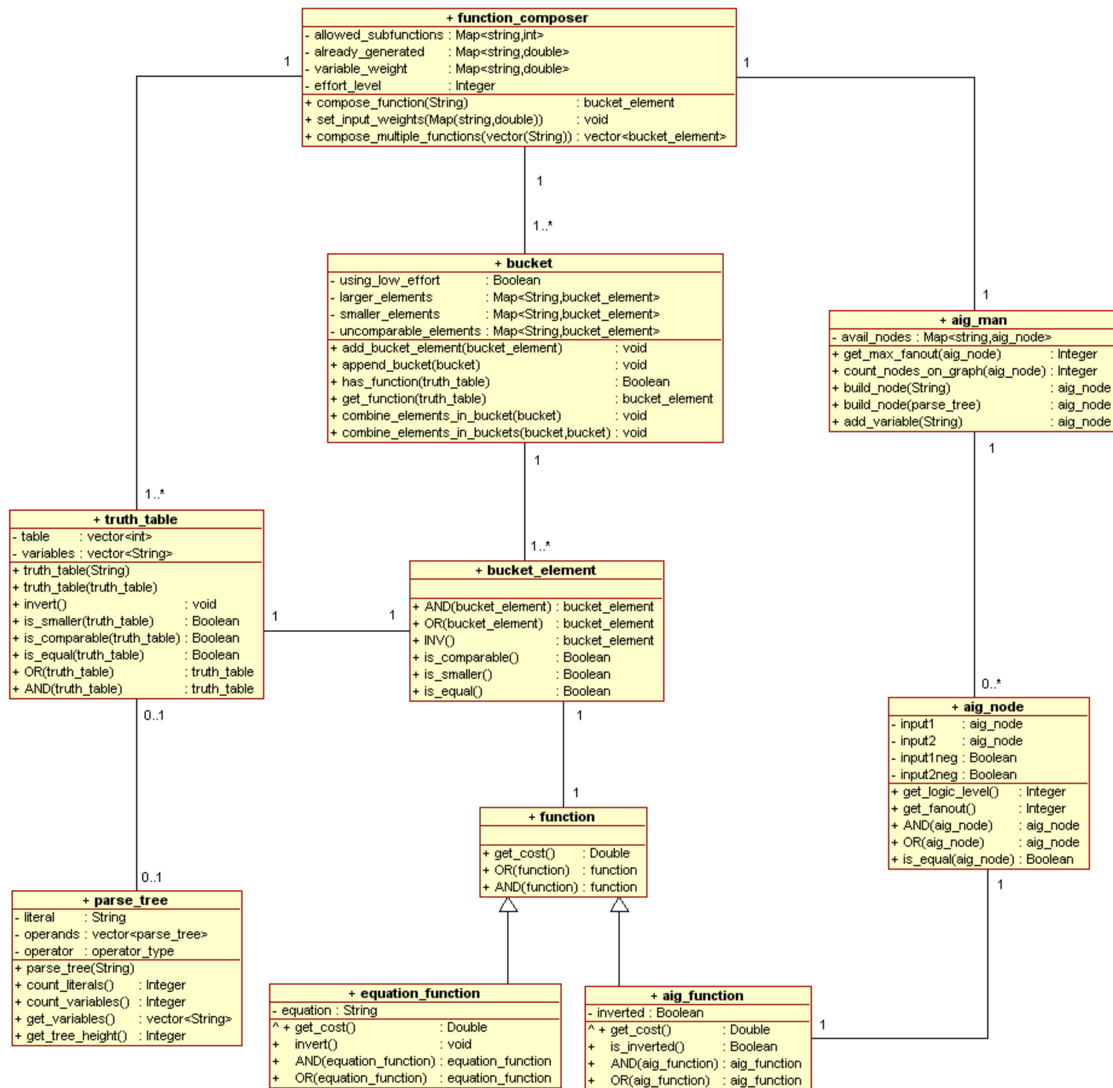



Figure 1. Class diagram of the implemented algorithm.

## AIG_Manager class

This class is responsible to handle the information common to all nodes of an AIG. Moreover, it builds the nodes and it keeps record of all available nodes to be instantiated and reused.

**Attributes**:

*vector<aig_nodes>* **avail_nodes**: Points to all nodes already created by the system that may be instantiated during the composition process.

**Methods**:

*integer* **get_max_fanout**(*aig_node node*): evaluates all the graph from the node given as parameter and retrieves the maximum fanout of the nodes of the subgraph.

*integer* **count_nodes_on_graph**(*aig_node node*): counts all nodes that are connected in the graph, using as root the node given as parameter.

*aig_node* **build_node**(*string equation*): all AIG nodes necessary to represent the function described by the equation provided as parameter and retrieves the root node of the graph.

*aig_node* **build_node**(*parse_tree pt*): builds all AIG nodes necessary to represent the function described by the parse_tree provided as parameter and retrieves the root node of the graph.

*aig_node* **add_variable**(*string variable*): adds to the manager an AIG node representing the variable whose name is given as parameter.

## AIG_Node class

This is the class responsible for storing all information pertinent to the nodes of an AIG. It also performs the composition of nodes using either AND or OR operations.

**Attributes**:

*aig_node* **input1**: One of the inputs of the node. It is always another AIG node since literals are considered a special case of an AIG node.

*aig_node* **input2**: Other of the inputs of the node. It is always another AIG node since literals are considered a special case of an AIG node.

*bool* **input1_negated**: Indicates the input1 is negated or not.

*bool* **input2_negated**: Indicates the input2 is negated or not.

*string* **id**: Identifier for an AIG node. In case the node is just a literal, it is the literal itself.

**Methods**:

*integer* **get_logic_level**(): returns the logic level (distance – in depth – of the more distant input) of the node.

*integer* **get_fanout**(): returns the fanout of the node, which is the number of elements that have it as a child node.

*aig_node* **AND**(*aig_node node*): performs the AND operation between a node given as parameter and the node itself. It creates and retrieves a new node, which has this node and the node given as parameter as child nodes.

*aig_node* **OR**(*aig_node node*): performs the OR operation between a node given as parameter and the node itself. It creates and retrieves a new node, which has this node and the node given as parameter as child nodes. It also performs the handling of the negation of the graph edges since it builds the OR operation using the AND node and negating the inputs and output by applying the DeMorgan law.

*bool* **is_equal**(*aig_node node*): compares two AIGs from the roots (one is the aig node itself and the other is the node provided as parameter) and returns if the complete graph is equal or not.

## Truth_Table class

This class stands for the truth tables of the system. It presents the output vector of the function it implements, it variables and the variable ordering of the vector.

**Attributes**:

*vector<int>* **table**: Output vector of the truth table. Each integer may represent from 32 to 64 bits of the truth table (depending of the system architecture).

*vector<string>* **variables**: The variables present in the function implemented by the truth table, in the same order that they appear on the table.

**Methods**:

**truth_table**(*string equation*): builds the truth table of the function described by the equation provided as parameter.

**truth_table**(*truth_table tt*): builds the truth table as a copy of the truth table provided as parameter.

*void* **invert**(): inverts all bits of the truth table output vector.

*bool* **is_smaller**(*truth_table tt*): compares two truth tables and returns if the truth table is smaller than the one provided as parameter.

*bool* **is_comparable**(*truth_table tt*): evaluates two truth tables and returns if they are comparable or not (being comparable is having the same number of bits and the same variables).

*bool* **is_equal**(*truth_table tt*): compares two truth tables and returns if they are equal or not.

*truth_table* **AND**(*truth_table tt*): performs the AND operation between a truth table given as parameter and the truth table itself. It creates and retrieves a new truth table.

*truth_table* **OR**(*truth_table tt*): performs the OR operation between a truth table given as parameter and the truth table itself. It creates and retrieves a new truth table.

## Parse_Tree class

This class is used to support functions described as equations. It handles an input equation and is used by the truth table class constructor when it receives an equation as input.

**Attributes**:

*string* **literal**: In cases where the parse tree is only one literal, the literal value is stored in this attribute. In other cases, this attribute is empty.

*vector<parse_tree>* **operands**: Vector containing one or two operands of this level of the parse tree (depending of the operation). These operands are also parse trees which allow the multi-level parse trees to be supported. The two operands are associated by the operation specified in the operator attribute.

*operator_type* **operator**: Indicates de operator used to associate the two operands of this parse tree root. Valid operators are “INV”, “AND” and “OR”.

**Methods**:

**parse_tree**(*string equation*): performs the parsing of the equation provided as parameter and builds the parse tree.

*integer* **count_literals**(): returns the number of literals present in the parse tree.

*integer* **count_variables**(): returns the number of variables present in the parse tree.

*integer* **get_tree_height**(): returns the height of the parse tree.

*vector<string>* **get_variables**(): retrieves all variables present in the parse tree.

## Bucket class

This class is responsible for handling all intermediate functions, grouping them according to their specific characteristics and combining them in order to generate new buckets.

**Attributes**:

*bool* **using_low_effort**: Indicates if the algorithm is running in disjoint effort level.

*map<string, bucket_element>* **larger_elements**: Stores all elements generated during this bucket construction which are larger then the target function.

*map<string, bucket_element>* **smaller_elements**: Stores all elements generated during this bucket construction which are smaller then the target function.

*map<string, bucket_element>* **other_elements**: Stores all elements generated during this bucket construction which are not comparable to the target function.

**Methods**:

*void* **add_bucket_element**(*bucket_element element*): adds an element to the bucket. When running in disjoint effort mode, it compares the element with the target function and stores it in the appropriate vector of elements (larger, smaller or other). When running in regular mode, all elements are added to the same vector (other).

*void* **append_bucket**(*bucket bkt*): adds all elements present in the bucket provided as argument to the bucket.

*bool* **has_function**(*truth_table tt*): searches the elements in the bucket trying to find the function provided as a truth table.

*bucket_element* **get_function**(*truth_table tt*): retrieves the element that corresponds to the function represented by the truth table provided as argument.

*void* **combine_elements_in_bucket**(*bucket bkt*): combines all elements in bucket among themselves in order to generate the elements that will be appended to the bucket.

*void* **combine_elements_in_buckets**(*bucket bkt1, bucket bkt2*): combines all elements in bucket 1 with all elements in bucket 2 in order to generate the elements that will be appended to the bucket.

## Bucket_Element class

This class stands for the elements that are handled by the bucket class. They represent a function and are associated to both a truth table (which is used as a key in a hash due to its canonicity) and a function, which may be an AIG (whose construction is the main purpose of this application) or an equation (for comparison and test reasons). The bucket element objects know how to combine and compare themselves.

**Attributes**:

*function* func: The function (either described as an AIG or as an equation) which is part of the bucket element.

*truth_table* tt: The truth table of the function of the bucket element.

**Methods**:

*bool* **is_smaller**(*bucket_element bkt*): compares two bucket elements and returns if the bucket element is smaller than the one provided as parameter.

*bool* **is_comparable**(*bucket_element bkt*): evaluates two bucket elements and returns if they are comparable or not (being comparable is having the same number of bits and the same variables in the truth table associated to the them).

*bool* **is_equal**(*bucket_element bkt*): compares two bucket elements and returns if they are equal or not. Two bucket elements are considered equal if both their truth tables and their function associated (either an equation or an AIG are equal).

*bucket_element* **AND**(*bucket_element bkt*): performs the AND operation between a bucket element given as parameter and the bucket element table itself. It creates and retrieves a new bucket element.

*bucket_element* **OR**(*bucket_element bkt*): performs the OR operation between a bucket element given as parameter and the bucket element itself. It creates and retrieves a new bucket element.

*bucket_element* **INV**(): inverts the bucket element (inverting all bits of the truth table and inverting the function) and retrieves the resulting new bucket element.

## Function class

This class represents every function of the system. It implements the methods and presents the attributes that are common both to the functions represented as AIGs and to the functions represented as equations.

## Equation_Function class

This class implements the functions that are represented by equations. It inherits from the function class and it may be associated to a bucket element if the system is operating over equations (and not AIGs).

**Attributes**:

*string* **equation**: string containing the equation of the function. For instance: “a+b*c”.

**Methods**:

*double* **get_cost**(): performs the cost estimation of the associated equation (mainly building a parse tree and obtaining its characteristics such as depth, number of literals and number of variables).

*void* **invert**(): inverts the equation.

*equation_function* **AND**(*equation_function func*): performs the AND operation between a equation function given as parameter and the equation function itself. It creates and retrieves an equation function.

*equation_function* **OR**(*equation_function func*): performs the OR operation between a equation function given as parameter and the equation function itself. It creates and retrieves a new equation function.

## AIG_Function class

This class implements the functions that are represented by AIGs. It inherits from the function class and it is associated both to the AIG manager class (which stores all AIG and to the AIG node class, which is the root of the function graph)

**Attributes**:

*bool* **inverted**: Flag that indicates if the output of the AIG associated to this object is inverted or not.

*AIG_node* **node**: The root node of the AIG associated to this object.

**Methods**:

*double* **get_cost**(): performs the cost estimation of the associated AIG.

*bool* **is_inverted**(): returns if the associated AIG output is inverted or not.

*aig_function* **AND**(*aig_function func*): performs the AND operation between a AIG function given as parameter and the AIG function itself. It creates and retrieves a AIG function.

*aig_function* **OR**(*aig_function func*): performs the OR operation between a AIG function given as parameter and the AIG function itself. It creates and retrieves a new AIG function.