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

WALTER LAU NETO

# Exact Multi-Level Benchmark Circuit Generation for Logic Synthesis Evaluation

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

Advisor: Prof. Dr. Renato Ribas

Porto Alegre
August 2018

**CIP — CATALOGING-IN-PUBLICATION**

Lau Neto, Walter

Exact Multi-Level Benchmark Circuit Generation for Logic Synthesis Evaluation / Walter Lau Neto. – Porto Alegre: PGMICRO da UFRGS, 2018.

69 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2018. Advisor: Renato Ribas.

1. Digital circuit design. 2. Logic synthesis. 3. Exact benchmarks. 4. Synthesis algorithm evaluation. 5. Reversible logic. I. Ribas, Renato. II. Título.

*"If you limit your actions in life to things that nobody can possibly find fault with, you will not do much!"*

— Lewis Carroll

## ACKNOWLEDGMENT

To my family, in special to my parents Richard and Claudia, my aunt Nelza and my grandmother Astrid thank you for supporting me in my studies. To my uncle Sérgio and my aunt Maria Candida thank you for receiving me in your home when I moved to Porto Alegre, in 2011. Thanks to that I could have the opportunity to come to study here. Also, I would like to thank my brother, Nicholas, who has tolerated to share an apartment with me in the last years.

To my advisor, Renato Ribas, thank you for the patience and the valuable lessons. In particular, I would like to thank you for the discussions about life in general. Even though the technical knowledge is essential, many times it may be found in books and, in our area, tends to become deteriorated in a short time. However, the kind of lessons I have learned from you about research and life will remain with me forever.

Also, I would like to thank professor André Reis for trusting on me and giving me opportunities to improve my knowledge. Furthermore, I would like to thank my colleagues (friends) from LogiCS labs. Thank you for receiving me so well. I am grateful for all of our discussions, funs, barbecues and soccer. In the scope of this work, I would like to thanks Felipe Marranghello, for discussing reversible logic with me from the beginning of my studies, as well as Vinicius Possani for always explaining me the concepts of logic synthesis and algorithms.

To my friends from PUCRS and GAPH thanks for the discussions and fun we have had during these years. Special thanks to Alexandre Amory who has introduced me to the research and to Matheus Moreira who has taught me a lot and is a great friend.

Finally, to all my friends thank you for understanding when I could not go somewhere because of my studies. Also, thanks for all the funny moments we have had.

## ABSTRACT

Electronic design automation (EDA) tools provide a highly automated flow for integrated circuit (IC) design. This flow may be roughly divided into three main steps: high-level synthesis, logic synthesis and physical synthesis. The logic synthesis step has as goal circuit logic optimization and circuit implementation in a given technology. Usually, the logic synthesis is performed over a multiple-level network, which implements the combinational logic of a given circuit. The problem of synthesizing a multi-level network is a complex task, where exact synthesis is just practical for functions with a few inputs, and the vast majority of algorithms are heuristic. While validating and evaluating new heuristic methods, benchmarks are of great importance. Usually, when a new method emerges, it is compared to the previous best-known results for a similar set of circuits, showing the relative efficiency of this new method over the previous one. However, with such an evaluation it is not possible to assess if current approaches are producing a near-optimal solution or if there is still room for improvement. To address this issue, it is of great interest have circuits with an exact known solution. In this work, a novel method to generate exact multi-level logic circuits is presented. The proposed method is based on reversible logic and creates circuits acting as the identity function $f(x) = x$. It means that the generated circuits can be reduced to wires, with no gate instantiation. The proposed approach can generate exact benchmark circuits with up to 40 millions of *AND-inverter graph* (AIG) nodes in a few seconds. Furthermore, with the proposed method, it is possible to derive exact circuits in two different ways: (i) from real designs and (ii) building synthetic circuits. Both approaches are discussed, and logic synthesis results are presented running the state-of-art academic tools and a commercial tool for each. From results, it is possible to note that the generated circuits are challenging to logic synthesis tools and that there is a gap between the solutions found by these tools and the optimal circuit implementation. Finally, we present and discuss the flexibility of the proposed method, and how it can be further explored and applied in areas other than logic synthesis.



**Keywords:** Digital circuit design. Logic synthesis. Exact benchmarks. Synthesis algorithm evaluation. Reversible logic.

**Geração de Circuitos Multi-Nível com Solução Exata para Avaliação de Ferramentas de Síntese Lógica**

## RESUMO

Ferramentas de automação de projetos eletrônicos (do inglês *EDA*) proporcionam um fluxo automatizado para o projeto de circuitos integrados. Este fluxo pode ser dividido basicamente em três principais etapas: síntese de alto nível, síntese lógica e síntese física. Durante a síntese lógica, os principais objetivos são a otimização lógica do circuito bem como sua implementação em uma tecnologia alvo. Normalmente, a síntese lógica ocorre em uma rede multinível que implementa a lógica combinacional de um dado circuito. A síntese de redes multinível é uma tarefa complexa, de forma que a sua síntese exata é possível apenas para funções com poucas entradas, e a maioria dos métodos são heurísticos. Para avaliar novos métodos heurísticos, circuitos *benchmark* são de fundamental importância. Esta avaliação apresenta a eficiência relativa desta nova proposta em relação a anterior. Porém, ao avaliar a eficiência relativa, não sabemos se os métodos atuais já estão produzindo soluções próximas do seu ótimo, ou se ainda podem ser melhorados. Para avaliar esta questão, *benchmarks* onde se conhece a solução ótima são fundamentais. Neste trabalho, é apresentado um novo método para gerar circuitos multinível com solução ótima conhecida. O método proposto baseia-se em lógica reversível e gera circuitos que implementam uma função identidade $f(x) = x$. Isto significa que os circuitos gerados podem ser reduzidos a simples conexões sem nenhuma instância de porta lógica. A abordagem proposta pode gerar circuitos com até 40 milhões de nodos AIG (AND-*inverter graph*) em poucos segundos. Além disto, com o método proposto pode-se gerar circuitos exatos de duas diferentes maneiras: (i) baseado em circuitos reais e (ii) construindo circuitos sintéticos. Ambas abordagens são discutidas e para cada uma apresenta-se resultados de síntese lógica usando ferramentas de código aberto do estado da arte e ferramentas comerciais. Pode-se notar que os circuitos gerados são capazes de desafiar as ferramentas de síntese lógica, havendo diferenças consideráveis entre os resultados achados por estas e a solução ótima esperada. Finalmente, apresentamos e discutimos algumas flexibilidades do método proposto, bem como possíveis aplicações do método em outras áreas além da síntese lógica.

**Palavras-chave:** Projeto de circuitos digitais. Síntese lógica. *Benchmarks* exatos. Avalição de algoritmos de síntese. Lógica reversível..

## LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| AIG | And-Inverter Graph |
| ASIC | Application Specific Integrated Circuit |
| BDD | Binary Decision Diagram |
| BLIF | Berkeley Interchange Format |
| CAD | Computer Aided Design |
| CEC | Combinational Equivalence Checking |
| CMOS | Complementary Metal-Oxide-Semiconductor |
| DAG | Directed Acyclic Graph |
| DSD | Disjoint-Support Decomposition |
| EDA | Electrical Design Automation |
| FPGA | Field-Programmable Gate Array |
| HDL | Hardware Description Language |
| IC | Integrated Circuit |
| ILB | Identity Logic Block |
| ISOP | Irredundant Sum-of-Products |
| LEKO | Logic Synthesis Examples with Known Optimal |
| LEKU | Logic Synthesis Examples with Known Upper Bounds |
| LUT | Look-up Table |
| MAJ | Majority Logic Function |
| MIG | Majority-Inververter Graph |
| PLA | Programmable Logic Array |
| POS | Product-of-Sums |
| QoR | Quality-of-Results |
| RTL | Register-Transfer Level |

| | |
|---|---|
| SOP | Sum-of-Products |
| VLSI | Very-Large Scale Integration |

## LIST OF FIGURES

## LIST OF TABLES

## CONTENTS

# 1 INTRODUCTION

Since the emergence of the first integrated circuit (IC), in 1958, the number of transistor per IC has doubled every 18 months (WESTE; HARRIS; BANERJEE, 2005). This prediction was firstly done by Gordon Moore. being well known as the Moore's law (MOORE, 1965). With such a growth in integration, a single chip has evolved from thousands to millions and then to billions of transistors, leading to the very large integration scale (VLSI) era.

The first ICs were handcrafted designed, through an approach known as *full-custom*. However, when the designs have evolved to millions of transistors, this approach has proved to be unsustainable (RABAEY; CHANDRAKASAN; NIKOLIC, 2002). Therefore, designers started to look for new ways to adequate it for automation. In this scenario, the *semi-custom* design style has emerged. The most common *semi-custom* style is the standard-cell one, which is based in pre-designed libraries of cells used to implement circuits. Those cells are pre-designed and pre-characterized sub-circuits used in tandem to build more complex ones. With those sub-circuits, electronic design automation (EDA) environments are able to provide a highly automated flow.

Usually, an EDA environment performs three main steps: *high level synthesis*, *logic synthesis* and *physical synthesis* (MICHELI, 1994). The *high-level synthesis* is responsible for transforming the circuit behavior described in a higher level of abstraction into a hardware format, like *register transfer level* (RTL), which implements the circuit behavior. *Logic synthesis* consists of circuit logic optimization as well as implements the circuit targeting a given technology, for instance, field-programmable gate arrays (FPGAs) and application specific integrated circuit (ASIC). Finally, the *physical synthesis* assigns physical resources to the mapped circuit, *i.e.*, basically the placement of logic elements and routing of internal interconnection wires.

This work focuses on the logic synthesis step. To implement the circuit behavior, it is possible to have, at least, two-levels of logic or multiple levels. The main advantages of two-level logic over multi-level one are *speed* and *simplicity*, since the solution space is restricted, making the network easier to design and implement (HACHTEL; SOMENZI, 2006). Though two-level logic has been used to deal with programmable logic devices, such as programmable logic arrays (PLA) (FLEISHER; MAISSEL, 1975), it has limited usefulness in current VLSI circuit and system design, mainly due to the fact that two-level logic tends to oresent a larger circuit area and power dissipation when comparing with

multi-level logic (HACHTEL; SOMENZI, 2006).

On the other hand, deal with multi-level logic, which represents the majority of the circuits designed in practice, is a harder task (HACHTEL; SOMENZI, 2006). That is because the potential of reusing logic increases the size of solution space when comparing to two-level logic optimization (MICHELI, 1994). Thus, multi-level logic can be formulated as a dynamic search problem in which its complexity relies on the fact that the entire set of decisions is unknown until the search begins. Therefore, the search space grows according to the network growing during synthesis process (ERNST, 2009).

## 1.1 Logic Synthesis

The logic synthesis may be divided into sequential and combinational synthesis (MICHELI, 1994). Sequential synthesis deals with the sequential elements on the design, *e.g.* registers, whereas combinational synthesis optimizes the existing logic operations between sequential elements. In this work, we focus on combinational synthesis.

The combinational synthesis of two-level logic is well explored due to its simplicity. Hence, several tools for two-level logic minimization are available (COUDERT, 1994)(MCGEER et al., 1993), and functions with up to hundreds of inputs and outputs can be optimally synthesized in terms of gate count (ERNST, 2009) (HACHTEL; SOMENZI, 2006).

Nowadays, combinational logic circuits are usually implemented through multi-level logic, which tends to be smaller and consumes less power than two-level topology (HACHTEL; SOMENZI, 2006). Furthermore, multi-level logic allows a higher degree of freedom while designing a logic circuit (MICHELI, 1994), and different structures may be chosen to implement the same functionality (HACHTEL; SOMENZI, 2006). Unfortunately, this great freedom comes at the price of high complexity for logic optimization. Such a high complexity makes exact synthesis algorithms not practical, either while considering the number of logic gates or design logic depth. Though recent efforts have been made towards the multi-level logic exact synthesis (SOEKEN et al., 2018)(SOEKEN et al., 2017), it is still constrained by functions with a few inputs, and the proposed methods may be suitable to use in tandem with larger non-exact logic synthesis algorithms. Therefore, the majority of current logic synthesis algorithms are heuristic. These algorithms must meet some constraint criteria while minimizing other costs functions as much as possible (ERNST, 2009).

Whether the synthesis algorithm is exact or not, its assessment is done through circuit benchmarking. These are sets of standardized circuits thought to impose challenges to EDA (HARLOW, 2000). Usually, when a new method emerges, it is compared to the previous best-known results for a similar set of circuits. Standard metrics for such a comparison are speed, effectiveness and quality-of-results (QoR). This kind of contrast shows the *relative efficiency* of a new method over the previous one (AMARÚ et al., 2017).

## 1.2 Motivation

If no large improvements are achieved from one method to another, the standard method of assessing algorithms through its relative efficiency leaves an open question: are current methods already producing near-optimal solutions or is there still room for further improvement? (CONG; MINKOVICH, 2007). In other words, there is a lack to measure the method *absolutely efficiency* (AMARÚ et al., 2017). In order to achieve this goal and to answer the question, it is of great interest have circuits with a known exact solution.

The first work addressing the design of exact circuits regarding logic synthesis was proposed in (CONG; MINKOVICH, 2007). In such a work, the authors designed by hand a small synthetic circuit with the known optimal solution after technology mapping in terms of the number of 4-input look-up tables (LUT-4). This circuit was designed and tuned to be as hard as possible for structural-based mappers (CONG; MINKOVICH, 2007). The optimal implementation of this circuit is given by a binate-cover technique available on the SIS tool, which can find optimal solutions for small circuits (SENTOVICH et al., 1992). Thus, the authors present a method to build layers of these small circuits, by repeating and connecting them in order to create more complex designs where exact synthesis techniques are computationally unfeasible.

Recently, in (AMARÚ et al., 2017), the authors have proposed a method based on balanced binary trees in order to build exact synthetic circuits. By assuming that each leaf node is a different input variable, the authors guarantee depth optimality. Moreover, tricks to break trivial features of the generated basic tree are discussed aiming to complicate the synthesis tools. With the optimal tree, different possibilities to build a sub-optimal circuit are presented, which will serve as input to the logic synthesis tool. The work presents circuits with up to 600,000 *AND-inverter graphs* (AIG) nodes, and assess the

lack between the optimization done by FPGA open-source synthesis tools and the exact solution in terms of the circuit logic depth.

## 1.3 Objectives

This work aims to propose a novel method to generate exact multi-level logic circuits in terms of logic gate count and logic depth. Moreover, the resulting circuit must be complex enough to challenge the state-of-art logic synthesis algorithms. To accomplish this objective, we present two different approaches in Chapter 4.1 and Chapter 4.3.

The proposed methods differ from previous work in different ways, outstanding the following features: (i) it provides proven-by-construction optimal solution and does not depends on other methods/tools to guarantee the circuit exactness, and (ii) it unlocks the possibility to generate exact multi-level logic circuits with structures computing functions found in real applications.

Besides that, the method is flexible and can stress synthesis algorithms in respect to a given function property. Moreover, the method can be combined with previous works in order to increase their complexity. Finally, the proposed approach is able to generate exact circuits that are dozens of times larger in terms of gate count (AIG nodes) than those presented previously.

## 1.4 Proposed Work

This work proposes a new method to generate benchmark circuits with a known exact solution. The proposed method generates identity logic blocks (ILB), which has known exact solution in both circuit area and logic depth. Since the ILB performs the identity function $f(x) = x$, the optimum solution is zero for both metrics. The method generates ILBs with up to 40 millions of AIG nodes into a few seconds.

To derive the ILBs, the proposed method stands on reversible logic (SAEEDI; MARKOV, 2013). Reversible logic implements reversible functions which maps each input pattern to a unique output pattern (and *vice-versa*). Thus, reversible functions perform bijective functions. By using reversible logic, our method can both generate exact synthetic circuits by construction and derive exact circuits from real designs. This last feature is possible thanks to embedded methods, which transform irreversible functions

into reversible ones (SOEKEN et al., 2016b)(ZULEHNER; WILLE, 2017).

Finally, since our method implements identity functions, it is flexible and may be used in tandem with the methods proposed in (CONG; MINKOVICH, 2007) and in (AMARÚ et al., 2017). In fact, the proposed method can be applied to any circuit aiming to impose further difficulties to EDA tools. In the same way, exact functions can be embedded in our ILB in order to check either the tool is able to find the function itself or not, as discussed in Chapters 5.3 and 5.4. The embedding of Boolean functions into ILB, with a given characteristic, is useful to look for algorithms flaws under a known scenario.

## 1.5 Text Organization

The rest of the text is organized as follows:

- In Chapter 2, some fundamentals on Boolean functions are given to introduce the concept of reversible functions that the proposed method implements. Moreover, it presents some important ways to represent Boolean functions, which are utilized along the work for both implementation and assessment. Finally, it reviews the IC design flow, where we focus on the logic synthesis evaluation.
- Chapter 3 presents related works on benchmarks. This chapter covers previously proposed benchmark circuit set.
- The proposed method to generate exact circuits is depicted in Chapter 4. Two approaches are presented: (i) generating exact circuits from real designs and (ii) generating synthetic exact circuits. Experimental results show that in both flows there is still room for improvements.
- In Chapter 5, synthesis results are presented for an open-source and commercial tool. Besides showing results for exact circuits derived from real designs and synthetic exact circuits, we also present variations of the proposed block in order to increase its complexity, within the synthesis results.
- Finally, Chapter 6 summarizes and concludes the contributions of this work.

# 2 PRELIMINARIES

This chapter presents the fundamentals for understanding this work. It discusses the notion of Boolean functions, reversible Boolean functions and some ways to represent such functions. Finally, it presents the IC design flow with a focus on logic synthesis for FPGAs and standard-cell based ASICs.

## 2.1 Boolean Function Definitions

The *Boolean set*, defined as B = {0,1}, represents two logic values, *i.e.*, *false* and *true*. A Boolean set of dimension $n$ ($B^n$) is composed by all distinct elements of length $n$. Thus, the $B^n$ set has $2^n$ elements, *e.g.* $B^0 = \emptyset$, $B^1 = \{0, 1\}$ and $B^2 = \{00, 01, 10, 11\}$.

An $n$-input Boolean function *f(X)* is a mapping $f: B^n \Rightarrow B$ defined by its input variables (support) X = $\{x_0, x_1, ..., x_{n-1}\}$, where the variables are in the Boolean domain, *i.e.*, each variable can only assume values of $B$. The function maps each of the $2^n$ elements from its domain set to either 0 or 1 into its image set.

A *multiple output Boolean function* is a mapping $f: B^n \Rightarrow B^m$, with $n, m \in \mathbb{N}$. In other words, it is a system of functions $f_i = \{x_0, x_1, ..., x_{n-1}\}$, with $0 \leq i \leq (m-1)$. In this work, multiple-output functions are denoted by $n$ x $m$ functions.

An $n$ x $m$ Boolean function is said to be reversible if, and only if, it maps each input pattern into a unique output pattern (and *vice-versa*), and $n = m$. Therefore, reversible functions realize bijective Boolean functions, *i.e.*, each element of the input set is paired with one element at the output set, and there are no unpaired elements. Since for reversible function $n = m$, the output set is a permutation of the input set. Figure 2.1 presents the input and output sets of a bijection function $f : X \Rightarrow Y$. Note that, for a given output, it is possible to determine its associated input pattern. That is, a bijection has an inverse mapping $f^{-1} : Y \Rightarrow X$.

The relationship between the domain and the image sets of a Boolean function is usually done by logic operations over the variables from X. There are some basic operators, such as: AND ($\wedge$), OR ($\vee$) and NOT ($\neg$ or !). The AND operator evaluates the function to 1 when all $n$ variables are 1. Otherwise, the function is evaluated to 0. The OR operator, in turn, evaluates to 0 when all variables are 0. Otherwise, the function is evaluated to 1. The NOT operator, also known as negation, performs over only a single variable and returns 0 if the variable is 1, and *vice-versa*.

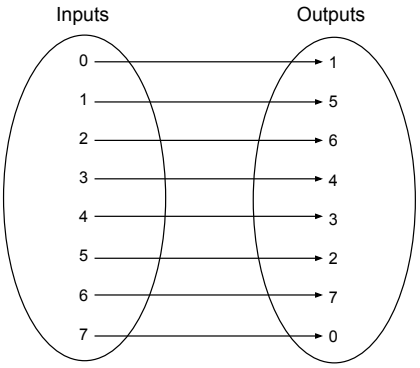



Figure 2.1: Input and output sets of a bijection $f : X \Rightarrow Y$.

## 2.2 Boolean Function Decomposition

Through decomposition, it is possible to express a complex Boolean function in terms of simpler subfunctions (ASHENHURST, 1957)(CURTIS, 1962). A given Boolean function $f(X)$ can be represented through the subfunctions $g$ and $h$, as follows:

$$f(x) = h(g(X1), X2) \tag{2.1}$$

with $X1$ and $X2$ different from $\emptyset$, and $(X1 \cup X2) = X$. This representation, when possible, is known as the functional decomposition of $f$. A special case of function decomposition is the *disjoint-support* decomposition (DSD), in which the sets $X1$ and $X2$ do not share any element, *i.e.,* $(X1 \cap X2) = \emptyset$. DSD can be treated by special algorithms (MISHCHENKO; BRAYTON, 2007), and has diversified applications in IC design domain (KUTZSCHEBAUCH; STOK, 2002)(PLAZA; BERTACCO, 2005)(BERTACCO; OLUKOTUN, 2002). The DSD is considered in some logic synthesis experiments carried out in this work.

## 2.3 Boolean Functions Representation

There are several ways to represent Boolean functions, differing in trade-off between simplicity and scalability. For instance, the truth table representation is a quite straightforward way to describe a given function, but it does not scale for large functions.

Therefore, this section presents an overview of models for representing Boolean functions used or referred to in this work.

### 2.3.1 Truth Table

Truth table is a widespread form to represent a Boolean function. For a function with $n$ inputs, its truth table is composed of $2^n$ rows and, consequently, by $2^n$ *minterms*, which are the product over the function variables. By applying the negation (NOT) operator over the minterm, it is possible to derive a maxterm. Therefore, through the De Morgan's theorem, maxterms are given as the sum over the function variables.

In truth table representation, for each possible combination of Boolean values assigned to the input variables, there is a correspondent output value. Table 2.1 shows the truth table of an arbitrary function $f(x_0, x_1, x_2)$, as well as its minterms. The minterms that evaluate $f$ to 1 form the function *on-set*. On the other hand, the minterms which evaluate $f$ to 0 are part of the function *off-set*.

Table 2.1: Truth table of arbitrary function.

| *x0* | *x1* | *x2* | $f$ | *minterm* |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $\overline{x_0}.\overline{x_1}.\overline{x_2}$ |
| 0 | 0 | 1 | 1 | $\overline{x_0}.\overline{x_1}.x_2$ |
| 0 | 1 | 0 | 1 | $\overline{x_0}.x_1.\overline{x_2}$ |
| 0 | 1 | 1 | 1 | $\overline{x_0}.x_1.x_2$ |
| 1 | 0 | 0 | 0 | $x_0.\overline{x_1}.\overline{x_2}$ |
| 1 | 0 | 1 | 0 | $x_0.\overline{x_1}.x_2$ |
| 1 | 1 | 0 | 1 | $x_0.x_1.\overline{x_2}$ |
| 1 | 1 | 1 | 0 | $x_0.x_1.x_2$ |

Alternatively, it is also possible to represent the function truth table $f(x_0, x_1, x_2)$ as a bit string, where the most significant bit refers to the last minterm (111) and the least significant bit refers to first minterm (000). For instance, the function presented in Table 2.1 can be represented as $f(x_0, x_1, x_2) = 01001110_2$.

The function presented in Table 2.1 has an output value for each of its input combination, and it is then said *completely specified*. Sometimes, however, it may be the case that under normal conditions, some input combinations should not occur. For those, the output value is said to be *don't care*, and both 0 or 1 can be assigned to the output without changing the function behavior. This choice is usually done looking for

the minimal synthesis solution. When there is an output function marked as *don't care*, the function is said to be *incompletely specified Boolean function*.

Reversible functions may be represented by truth tables with $2^n$ rows and an equal number of input and output columns. Since the most of truth-table based synthesis methods for reversible functions takes a completely specified function as input (WILLE; DRECHSLER, 2010), *don't* cares must be assigned while taking care to keep the bijective function feature.

Table 2.2 presents the truth table of an arbitrary reversible function with three inputs and three outputs. From this table, it is easy to note that there is one-to-one correspondence between inputs and outputs. Therefore, the function is bijective. A further discussion on embedding irreversible functions into reversible ones is presented in Chapter 3.

Table 2.2: Truth table of arbitrary reversible function.

| *x0* | *x1* | *x2* | *f0* | *f1* | *f2* |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 |

### 2.3.2 Sum-of-Products

Sum-of-products (SOP) form is an example of two-level logic expression, with cubes (minterms) joined by disjunction operators. Conversely, a function can also be represented through the product of its maxterms, which is known as product-of-sums (POS). If an SOP (POS) contains all the minterms (maxterms) on the function *on-set* (*off-set*), then it is canonical (unique). However, it is usually of interest to represent the SOP (POS) of a given function with the minimal number of cubes (sum terms). This work focuses on SOP representation.

The canonical SOP representation of the function presented in Table 2.1 is the following:

$$F = \overline{x_0}.\overline{x_1}.x_2 + \overline{x_0}.x_1.\overline{x_2} + \overline{x_0}.x_1.x_2 + x_0.x_1.\overline{x_2}. \tag{2.2}$$

On the other hand, the canonical POS representation of the same function is given by

$$F = x_0 + x_1 + x_2.\overline{x_0} + x_1 + x_2.\overline{x_0} + x_1 + \overline{x_2}.\overline{x_0} + \overline{x_1} + \overline{x_2}. \tag{2.3}$$

A cube composed of variables $x_0, ..., x_{n-1}$ is said to be *implicant* of $f(X)$, if any assignment that evaluates the cube to 1 does not map $f(X)$ to 0. An implicant cube can cover one or more minterm. A cube is *prime implicant* if it is not contained (covered) in any other implicant. *Essential prime implicants* are cubes that cover at least one minterm which is not covered by any other prime implicant.

If a set of cubes $S$ covers all the minterms of a function $f(X)$, so it also covers the function. The set $S$ is a *prime cover* of $f(X)$ if all its cubes are prime implicants. A *prime cover* of $f(X)$, where no prime cube can be removed from $S$ without changing the function behaviour, leads to an *irredundant sum-of-products* (ISOP). Since SOP and POS are dual, it is also possible to obtain an *irredundant product-of-sums* (IPOS) (BRAYTON et al., 1984).

The support variables $X$ of $f(X)$ can be represented in both positive and negative polarities. A literal is an instance of a variable $x_n$ or its complement $!x_n$. A cube is a conjunction (product) of literals. The weight of a cube is given by the number of literal it contains. By removing a literal, one doubles the number of input assignments that satisfy a given cube.

### 2.3.3 Programmable Logic Array

The programmable logic array (PLA) format was thought to describe circuits implemented in the PLA technology. This technology is comprised of a configurable architecture, which can be programmed to compute combinational logic. The architecture is composed by an array of AND logic gates, connected to the circuit inputs, followed by an array of OR logic gates, connected to the circuit outputs. In PLA architecture, both arrays are programmable differing from other architectures, such as PAL where only the AND array is configurable. Since PLAs are composed by AND logic gates followed by

Figure 2.2: PLA format representation of the function described in Table 2.1

```
# number of inputs
.i 3
# number of outputs
.o 1
# input signals name
.ilb x0 x1 x2
# output signals name
.ob f0
# on-set
001 1
010 1
011 1
110 1
.e
```

OR logic gates, they implement sum-of-product expressions. Figure 2.3 shows the PLA implementation of the same function presented in Table 2.1. There are $2^n$ AND logic gates in the AND plane, one for each minterm, as well as $m$ OR logic gates, where $m$ is the number of outputs of a given function. Each input can be connected to the AND logic gate directly or complemented through the inverter connected to each of them. The cubes that compose the function *on-set* are marked by a black dot and are connected to the OR logic gate.

Therefore, the PLA format representing a function implemented in such a technology is composed by the cubes that are in the function *on-set*. The Listing 2.2 shows the PLA file representing the circuit configuration in Figure 2.3. It is also possible to provide the *on-set* and *off-set* in the file.

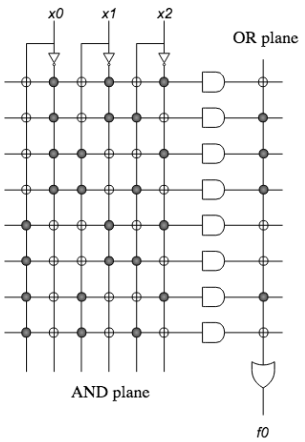



Figure 2.3: PLA architecture configurantion for the function presented in Table 2.1.

### 2.3.4 Berkeley Logic Interchange Format

The Berkeley logic interchange format (BLIF) is a textual format to describe hierarchical circuits. The BLIF format allows to describe the design internal nodes and is suitable for multi-level logic representation. The Listing 2.4 presents the BLIF description of the same circuit represented in the Table 2.1. In this code, *n5* and *n6* denote intermediate nodes as well as the cubes that evaluates them to true. The node *n7* is connected to the inverted outputs of nodes *n5* and *n6*. Finally, the primary output *f0* receives the negation of the node *n7* output. The same circuit is presented in Figure 2.6.

Figure 2.4: BLIF file describing the function presented in Table 2.1

```
# network name
.model blif\_example
# inputs
.inputs x0 x1 x2
# outputs
.outputs f0
# network connection
.names x0 x1 x2 n5
-10 1
.names x0 x1 x2 n6
0-1 1
.names n5 n6 n7
00 1
.names n7 f0
0 1
.end
```

### 2.3.5 Binary Decision Diagrams

A binary decision diagram (BDD) is a rooted, directed acyclic graph used to represent Boolean functions (AKERS, 1978). The graph vertex set $V$ accepts three types of vertices. A terminal vertex $v$ may denote a *true* or *false* decision, through the values 1 and 0, respectively. On the other hand, a *non-terminal* vertex $v$ has as attribute an input variable $x_i$ and represents a decision node with two children. If $x_i = 1$, go to *high(v)*, else go to *low(v)*. Finally, there is the function node, which has one incoming edge and no outgoing edges, and denotes the function being represented. Figure 2.5 presents the BDD representation of the function described in Table 2.1, respecting the variable order

$x_0 < x_1 < x_2$. The $F$ node denotes the function under representation. The dashed edges refer to the *low(v)* nodes.

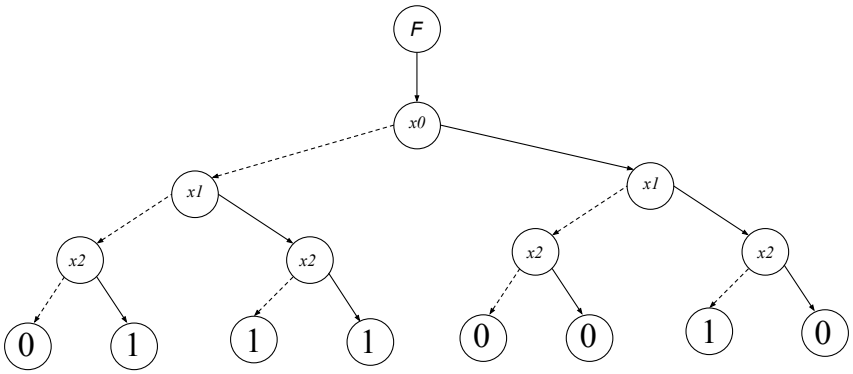



Figure 2.5: BDD representation of the function described in Table 2.1.

In order to make the BDD a canonical representation, the idea of reduced and ordered binary decision diagram (ROBDD) was introduced in (BRYANT, 1986), being a more compact way to represent BDDs. The canonicity is related to a given order of variables in the graph, which has a direct impact on the ROBDD size (HACHTEL; SOMENZI, 2006). In fact, for a given order, it may be the case that it is not possible to derive an ROBDD. For a given variable ordering, *non-terminal* nodes controlled by the same variable and pointing to the same child in both values ($x_i = 1$ and $x_i = 0$) are removed. Furthermore, nodes controlled by the same variable and pointing to the same left and right child are merged.

### 2.3.6 AND-Inverter Graph

AND-inverter graph (AIG) is the most common data structure for performing logic synthesis, even though it was first proposed to perform combinational equivalence checking (KUEHLMANN; KROHM, 1997). An AIG is a directed acyclic graph (DAG) used to represent Boolean functions, and its nodes have zero or two incoming edges. Nodes with zero incoming edges are *primary inputs* (PI), and nodes with two incoming edges represent the 2-input AND (AND2) logic operator. Also, nodes can be marked to represent *primary outputs* (PO). The operators may or may not be inverted. The inversion is represented by complementing the graph edges.

While using AIGs for logic synthesis, it is very common to compute *k-cuts* (PAN; LIN, 1998)(MISHCHENKO; CHATTERJEE; BRAYTON, 2007a). For a given AIG node

$n$, its cut $C$ is a set of nodes in the network, also known as leaves of the cut, such that every path from a PI to $n$ contains at least one node in $C$. A cut is said to be *k-feasible* if it contains no more than $k$ nodes. Otherwise, the cut is discarded. Usually, the *k-cuts* are computed into a single pass from PIs to POs, and the computation is performed as follows. If the node is a PI, it has a *trivial cut*, *i.e.*, the node itself is the cut. If a node represents the AND2 logic operator, its *k-cuts* is given by the cartesian product between the cut sets at each of its inputs as well as its *trivial cut*.

Figure 2.6 presents the AIG of the function represented in Table 2.1, as well as *k-cuts* for each node. The dashed lines represent inverters.

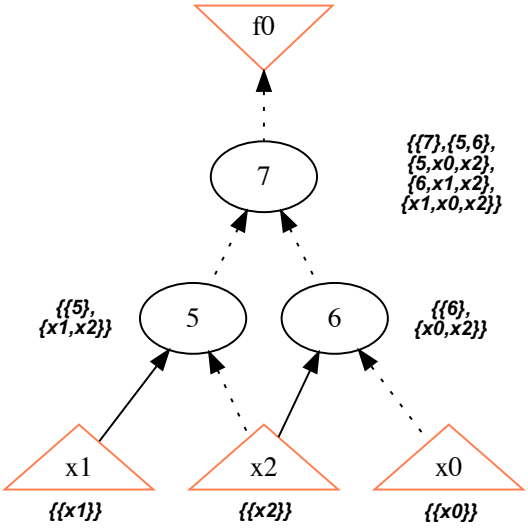



Figure 2.6: AIG representation of the function described in Table 2.1.

### 2.3.7 Majority-Inverter Graph

Majority-inverter graph (MIG) is a DAG which was recently proposed to manipulate Boolean functions (AMARÚ; GAILLARDON; MICHELI, 2014). In this model, each node has three incoming edges and represents the majority (MAJ) logic function. The MAJ function with $n$-inputs, being $n$ an odd number, returns true when at least $(\left\lceil \frac{n}{2} \right\rceil + 1)$ inputs are true. Similarly to AIGs, inversion on operators is represented through a complemented edge.

MIGs are a universal representation form, since AIGs are universal (BRAYTON; MISHCHENKO, 2010), and the majority node can implement the AND logic by setting

one of its inputs to 0. Therefore, MIGs $\supset$ AIGs. To unlock the MIG manipulation, in (AMARÚ; GAILLARDON; MICHELI, 2014), the authors present a Boolean algebra based exclusively on MAJ and inverter operations. This new algebra is composed of five fundamental transformation rules, which enables to explore the entire MIG representation space. The results presented by MIGs are promising and show improvements for signal delay propagation, circuit area and power dissipation when comparing to AIGs over the MCNC benchmark suite (AMARÚ; GAILLARDON; MICHELI, 2014).

Figure 2.7 shows the MIG representation of the same function represented by the AIG shown in Figure 2.6. Since that is a simple function, it is not possible to note any improvement in respect to its AIG representation. However, it is possible to note the MIG capability of representing the AND2 logic node by setting one of node input in 0. Therefore, the other two inputs must be 1 so that the gate output evaluates to 1.

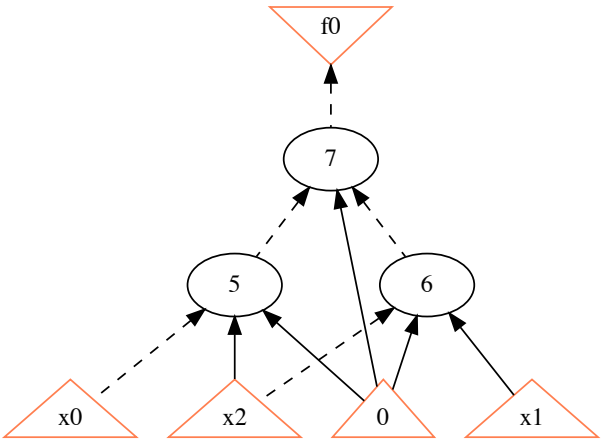



Figure 2.7: MIG representation of the function described in Table 2.1.

## 2.4 Integrated Circuit Design

As discussed before, the role of EDA is to provide an automated flow for designing ICs. Such a flow relies on the *semi-custom* design style, which has a shorter design time when comparing to the *full custom* approach, at the price of a larger penalty in performance and power consumption. This work concerns the design flows for field programmable gate array (FPGA), as well as the standard-cell approach for application specific integrated circuit (ASIC) technology.

There are three main steps performed by the EDA tool: *high level synthesis*, *logic synthesis* and *physical synthesis* (MICHELI, 1994). In this work, we focus on the logic

synthesis step, which has two primary goals: (i) to transform a digital circuit described in a high level of abstraction, usually in hardware description language (HDL), into a logic network (AIG/MIG), and (ii) to optimize the resulting logic network. While performing logic synthesis over the design, the tool has multiple goals, such as minimize circuit area, power consumption and critical path delay.

The logic network optimization is performed into two steps. The first one is known as technology independent optimization and consists of optimizations such as constant propagation, logic sharing, redundancy removal and circuit restructuring (CHEN et al., 2006). As a second step, the technology independent logic network is mapped into *look-up tables* (LUT) for FPGA or standard cells, when considering ASIC. The technology mapping has a significant impact in the final physical implementation and is subject of extensive work (CHEN et al., 2006). Technology mapping methods can be classified according to their optimization goals (area, power, timing, routability) or according to the transformation techniques explored while mapping, *i.e.,* structural or functional (CHEN et al., 2006). This work studies mapping methods in respect to their transformation techniques.

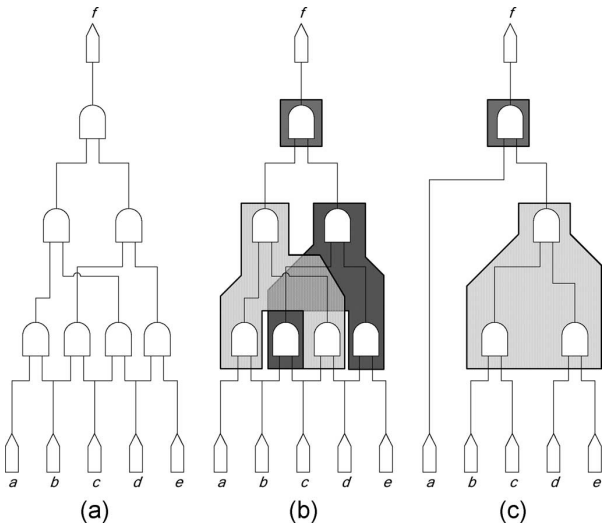



Figure 2.8: (a) Original circuit. (b) Structurally mapped. (c) Functionally mapped.(CONG; MINKOVICH, 2007)

Structural mappers are those in which the logic network is not modified and the mapping may be seen as covering problem, where sub-graphs from the logic network are mapped to standard cells or LUTs. Even though this method is suitable for large designs (CHEN et al., 2006), it depends strongly on the input logic network, which is known as structural bias (CHATTERJEE et al., 2006).

On the other hand, functional mappers are those that mix Boolean optimization with covering. Since functional mappers perform Boolean optimization and transformations over the input logic network, it can explore a larger solution space than the structural mapper, at the price of being time-consuming (CHEN et al., 2006) (MISHCHENKO; CHATTERJEE; BRAYTON, 2007b).

Figure 2.8, from (CONG; MINKOVICH, 2007), presents the difference from functional and structural mappers. The left-hand image presents the original circuit before mapping. This circuit realizes the AND operation of all its inputs, and each AND logic gate can be seen as an AIG node. While mapping this circuit, the structural mapper role is just to cover the logic network (subject graph), mapping the nodes to LUTs, as presented in Figure 2.8(b). The functional mapper, in contrast, can minimize the Boolean function while mapping, resulting in just one 4-input LUT (LUT4), presented in Figure 2.8(c).

# 3 RELATED WORK

The complexity of the multi-level logic synthesis problem, where optimal (exact) solutions are impractical for current real size designs, pushes the electronic design automation (EDA) community to adopt benchmark circuits in order to evaluate and compare those methods. While a diversified set of benchmark suites have been proposed so far, the exact solution is unknown for most of them. Therefore, it is quite difficult for developers to estimate the efficiency of their tools and possible improvements. In this context, exact benchmark circuits are of paramount importance since they allow the evaluation of synthesis methods and algorithms concerning the exact expected solution. This chapter first presents general benchmark circuits as well as their main characteristics. Afterwards, we present and discuss exact benchmark circuits, highlighting their features.

## 3.1 Benchmark Circuits

Benchmarks are standards by which similar things can be compared against the same reference. In the circuit design scope, benchmarks are standardized circuits which impose challenges for algorithms (tools). These circuits are used to compare performance among different algorithms, being standard metrics for such a comparison the speed, effectiveness and quality-of-results (QoR). Furthermore, benchmarks can end up by finding out problems that algorithms can not deal at all. When it happens, they unlock the possibility of diagnosing the algorithm flaw and may lead to innovation (DAVIDSON; HARLOW, 2000).

Back to '60s and '70s, the first benchmark circuits were freely available in the form of a data book provided by Texas Instruments, and anyone could look at it. Such a databook was composed by large-scale integration (LSI) gate-level designs (DAVIDSON; HARLOW, 2000), such as arithmetic logic unit (ALU) and counters. However, through the years, designs have become more complex and with a higher cost, making it difficult for universities acquiring industrial grade circuits. At this time, the lack of available complex benchmark circuits to universities has precluded the emerging of novel techniques. Therefore, the efforts were concerned with incremental improvement of existing methods (DAVIDSON; HARLOW, 2000).

In this context, the first effort regarding a free standardized set of circuits was made by Franc Brglez and Hideo Fujiwara. This set was composed of ten purely combinational

circuits and was published in a special issue at the IEEE International Symposium on Circuits and Systems (ISCAS), in 1985. Nowadays, this set is still trendy and is well known as ISCAS'85 (BRGLEZ; FUJIWARA, 1985). In its first release, these circuits were described in a proprietary logic format, and a translator written in FORTRAN was distributed along with the set of circuits, to convert them into other popular formats (DAVIDSON; HARLOW, 2000). In this benchmark suite, the combinational circuits are multi-level, and its primary purpose when released was to assess combinational automatic test-pattern generation (ATPG) tools. Despite that, the ISCAS'85 circuits have been used to evaluate methods in additional areas, including logic synthesis. Even though these circuits are very small and ceased to be good representatives for the majority of current applications, they are still widely adopted in academia. This work has had a considerable impact, leading to an increase in the test generation research (DAVIDSON; HARLOW, 2000).

From this first effort, a trend has evolved and there were several workshops, special sessions and scientific articles introducing new and/or modified circuits for benchmarking, targeting different areas in circuit design. In 1989, the ISCAS'85 suite was extended and sequential circuits were added up as well as more complex circuits. This has led to the ISCAS'89 benchmark suite, with 31 new sequential circuits. At the time, this new set of circuits has increased the work on sequential ATPG tools (DAVIDSON; HARLOW, 2000). However, such as the ISCAS'85, this suite is still very used but its complexity is too low for new challenges. For instance, the most complex circuit consists of 22,179 gates and 1,636 flip-flops (BRGLEZ; BRYAN; KOZMINSKI, 1989), where it is expected trillions of logic gates to design future generation of ICs (STOK, 2013).

Also in 1989, at the International Workshop on Logic and Synthesis (IWLS), the Microelectronics Center of North Carolina (MCNC) has introduced a new set of circuits for benchmarking (YANG, 1991b). This new set was composed by 2-level logic circuits, which were firstly introduced by U. Leuven, in 1985, and by small synthetic circuits. Later, the set introduced by U. Leuven was extended by a group from Berkeley, with some industrial circuits and arithmetic operands. The junction of these two sets, with some industrial multi-level and finite state machine circuits, as well as the ISCAS'85 and ISCAS'89 suite, has led to the MCNC benchmark suite (YANG, 1989). The MCNC circuit set has later become known as the LGSynth'89 benchmark suite, which was primarily proposed for logic synthesis and optimization. In IWLS'91 and IWLS'93, LGSynth'89 was successively extended to LGSynth'91 (YANG, 1991a) and LGSynth'93 (MCELVAIN,

1993), respectively. From these efforts, different conferences and workshops followed this trend by publishing new benchmark suites, including HLSynth92 and PDWorkshop93, just to name a few (DAVIDSON; HARLOW, 2000). Although it is still being used in some academic works, they no longer represent challenges for most of the applications nowadays. Its main issue is the size of the circuits.

In the 1990's, industrial circuits have become more complex, and there was a gap between publicly available benchmarks and real designs. In this context, in 1999, at the International Test Conference (ITC), was published the ITC'99 benchmark suit. The ITC'99 suite was extended in 2000 and it was firstly designed to increase the complexity from previous benchmarks, as well as to provide realistic circuits as study cases, aiming research in design-for-testability (DFT) and ATPG (CORNO; REORDA; SQUILLERO, 2000) (BASTO, 2000). The ITC'99 suite has four branches, as follows:

- I99X subset comprises ASIC and ICs designs and has fairly good-sized industrial circuits, including superscalar microprocessor and digital signal processor (DSP) circuits (DAVIDSON; HARLOW, 2000).
- I99T is a subset of Politecnico di Torino which comprises 22 diversified circuits, namely from b01 to b22. This branch includes from simple circuits, such as finite state machines (FSM), to a pseudo-system-on-chip (SOC). The first goal of this branch is to enable researchers to develop algorithms working on register transfer level (RTL) circuits described in VHDL (CORNO; REORDA; SQUILLERO, 2000).
- I99S presents some circuits from ISCAS'89 described in RTL. Even though they keep the same flaws from the original ISCAS'89 subset, *i.e., low complexity*, their goal is to provide a way to compare RTL level *versus* gate level ATPG.
- I99C is a special circuit derived from industrial design and it is said to cause troubles in industrial ATPG approaches (Scott Davidson, 1999).

Even though the ITC'99 has accomplished its initial purpose and is composed by reasonably good-sized circuits (up to 98K+ gates and 6K+ flip-flops), this benchmark suit lacks from maintenance. Except for the I99T subset, all other circuits are hard to find for download. Even for the I99T subset, its original function may have been lost during the development process. Furthermore, its release clearly states that, due to the development process, there is no guarantee that VHDL descriptions are functionally meaningful (CORNO; REORDA; SQUILLERO, 2000).

By the year 2005, a new benchmarking effort has been made by the IWLS com-

munity. The IWLS'05 benchmark is a joint of previously proposed suites, *i.e.,* ISCAS'85, ISCAS'89 and ITC'99 (subset I99T), with three new set of circuits (ALBRECHT, 2005). One of them is a selection from the OpenCores whereas the others are industrial initiatives, from Gaisler and Faraday. The Gaisler benchmarks are composed by the LEON2 processor, developed by and to the European Space Agency (ESA), and by LEON3, compliant with the SPARC V8 architecture. From the Farady side, three functional blocks were provided, a 16 bit DSP with SRAM blocks, a 32-bits RISC CPU and a direct memory access (DMA) controller. This benchmark is available in two formats, *i.e.,* OpenAccess and Verilog, and in total has 84 good-sized circuits, with up to 900,000 cells and 185,000 registers. The IWLS'05 suite is well maintained and is a good representative for many of current challenges.

In 2007, the Altera Corp. and UC Berkeley released a new FPGA-oriented benchmark suite at IWLS (PISTORIUS et al., 2007). The initial release consisted of eight large designs, each comprising at least 10,000 4-input look-up-tables (LUT-4) and, in addition to the distribution of the circuits, it has the goal to provide a reference compilation flow and assess logic synthesis and technology mapping algorithms available in EDA environments. The flow supports designs written in VHDL, Verilog and SystemVerilog. Moreover, this benchmarking effort has been updated and an interesting improved subset claims attention: 12 medium-size OpenCores designs free of multi-entity hierarchies, memories or other hard blocks (which would create design flow restrictions), and also free of adders/multipliers (which would be functionality synthesized to macros in ASIC flows or mapped to dedicated circuitry in FPGAs). The main drawback of this set is also the size of the circuits.

Recently, in 2015, a new set of benchmarking circuits was introduced, known as EPFL benchmark suite (AMARÚ; GAILLARDON; MICHELI, 2015). This set was published in IWLS'15 and is composed by purely combinational circuits. The set includes 10 arithmetic circuits, 10 random/control circuits and 3 synthetic circuits with *more than ten million* (MtM) gates, ranging from 16 to 23 millions of nodes. The arithmetic part includes from simple circuits, such as an adder, to more complex circuits, as the hypotenuse operator. The same is true for the random/control set, which varies from an ALU controller to a memory controller. This initiative disposes of these circuits in different formats and keeps track of the best-reported results online to serve as a reference for the community.

Table 3.1 presents a summary of the benchmarks described in this section. The

table presents their complexity, first purpose when released and the maximum number of gates (or AIG nodes) reported. It is possible to note that the efforts to design new and more complex circuits for benchmarking is a continuous process.

Table 3.1: Summary of presented benchmark sets.

| Name | Year | Logic Levels | Complexity | Purpose | Max Number of Gates |
|---|---|---|---|---|---|
| ISCAS | 1985 | multi | low | combinational ATPG | 3,512 cells |
| ISCAS | 1989 | multi | low | sequential ATPG | 22,179 cells |
| MCNC | 1989 | 2 and multi | low | logic synthesis | ∼35,000 cells |
| ITC | 1999 | multi | medium | ATPG | ∼98,000 cells |
| IWLS | 2005 | multi | complex | logic synthesis | ∼900,000 cells |
| EPFL | 2015 | multi | complex | logic synthesis | ∼23 millions of AIG nodes |

Although these benchmarks present a variety of circuits, complexity and are still widely adopted, they all lack from an important feature. From these sets of circuits, it is possible to evaluate new algorithms and tools with respect to the previous best-known results. It means, it is possible to assess a new algorithm in respect to another for some metric, which shows its *relative efficiency* (AMARÚ et al., 2017). Though a vast improvement from previously reported result could be achieved, it is still hard to know whether the algorithm can be further improved, *i.e.*, there is room for more optimization or the method has found the optimum circuit implementation. To do such an evaluation, it would be necessary to have a set of circuits with a known exact solution, so that it would be possible to assess the *absolute efficiency* of algorithms.

## 3.2 Exact Benchmark Circuits

An optimality study was firstly proposed inside the placement community, which had faced a slow down in research effort in the 2000s. Therefore, the question that had emerged at the time was if the placement methods had hit a plateau. In order to look for this answer, in (CHANG et al., 2004), the authors proposed the called *placement examples with known optimal* (PEKO). In this study, they presented that the state-of-art tools at the time produced wirelengths ranging from 1.66 to 2.53 times the optimal solution. This work renewed the interest of this community to look for new methods and, in three years, the optimality gap on the PEKO was reduced to around 20% (CHANG et al., 2004).

In that context, the first study with respect to circuits with exact solution aiming to evaluate logic synthesis algorithms was proposed in (CONG; MINKOVICH, 2007). In

this work, the authors propose a study where the optimal solution after FPGA technology mapping is known. The primary motivation is that throughout the 1990s, several works were published regarding FPGA synthesis and technology mapping, but a decrease in those publications have taken place during the 2000s (CONG; MINKOVICH, 2007). Therefore, to investigate whether proposed methods have hit a plateau and were near-optimal solutions, the author proposed an optimality study, called *logic synthesis examples with known optimal* (LEKO).

The design of the LEKO circuits is based on the design of small core circuit. The core circuit presented in (CONG; MINKOVICH, 2007) was handcrafted to be as hard as possible for structural mappers. The designed core circuit must have the following features:

- the core circuit has the same number of inputs and outputs;
- each output is a function of all inputs;
- each internal node of the core circuit has exactly two inputs; and
- there is an exact mapping solution in terms of LUT-4.

To proof the exact solution of the core circuit presented in their work, the authors rely on a command of binate-covering available on SIS tool (SENTOVICH et al., 1992). This algorithm can find exact mapping, in terms of LUT-4, for circuits with up to a hundred of logic gates. To generate huge circuits, so that the algorithm cannot find the exact solution, is proposed a method for stacking the core circuit in layers, such that by the end there is only one way to traverse the resulting graph from the outputs to get to the inputs. Since the exact solution of the core circuit is given in terms of LUT-4, the LEKO known optimal solution is in respect to the circuit area.

Let $C_5$ denotes a core circuit with five inputs and outputs, as presented in Figure 3.2. To design a LEKO circuit with $L$ layers, the first step is to create a bottom layer with $n^{L-1}C_n$. Then, for each layer, one should make $n^{L-1}$ copies of the $C_n$ and connect the outputs from the previous layer to the inputs of the new layer. The connections must be spread in a way that each $C_n$ in the top level is connected to every $C_n$ at the bottom level. To design larger circuits, one must increase the number of layers. Figure 3.1 shows a LEKO with two layers of $C_5$ circuits. With this scheme, the authors show that the best solution for mapping each layer of the LEKO is given by mapping each $C_n$ optimally and separately, and not by mapping between layers.

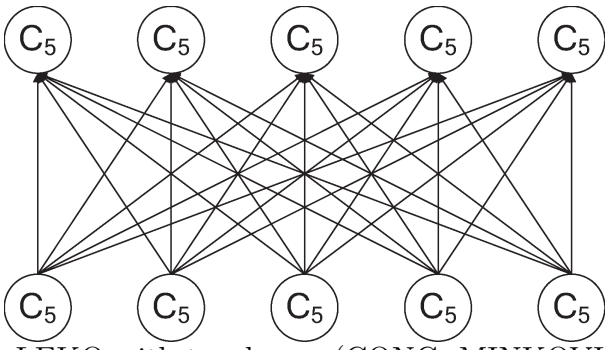



Figure 3.1: LEKO with two layers (CONG; MINKOVICH, 2007).

In the same work, the authors have also introduced the *logic synthesis examples with known upper bounds* (LEKU). The LEKU circuit is generated from the LEKO one by firstly collapsing the circuit in a 2-level network, and then decomposing the resulting network into an equivalent one, bounded to 2-input simple gates. The network collapsing and decomposition are done by SIS tool (SENTOVICH et al., 1992). By executing these steps, redundant logic is added to the original LEKO circuit, deriving the upper bound solution, since both LEKO and LEKU are functionally equivalent. The results show that, at the time for the LEKO case, the state-of-art mappers were far from finding the optimal solution. For the LEKU case, industrial and academic FPGA synthesis flows, both were around 70 times larger in terms of area in average and up to 500 times larger in the worst case.

More recently, in (AMARÚ et al., 2017), Amarù *et al.* have proposed a new constructive approach for exact multi-level logic circuit generation. Since the LEKO circuits had an exact solution in respect to the number of LUTs, the authors have proposed a new method to synthesize circuits with known optimal logic depth. It is of interest to evaluate how current algorithms perform while optimizing for depth since it correlates with the delay of the final system.

For designing exact circuits, the approach relies on balanced binary trees. The input of the method is the level of complexity targeted. For a level $n$, the resulting tree has $2^n$ input variables. While constructing the balanced binary tree, each node is randomly assigned for one among ten binary Boolean operators, each one depending on two variables (AMARÚ et al., 2017). Furthermore, each leaf of the tree represents a separate variable, so that the resulting tree is depth-optimal. Figure 3.3 presents a possible generated balanced binary tree of complexity two realizing the function $f = abc + abd$. This representation is depth optimal because a balanced binary tree of depth $n$ depends on all its $2^n$ leaf variables. Suppose it was possible to implement the function in Figure

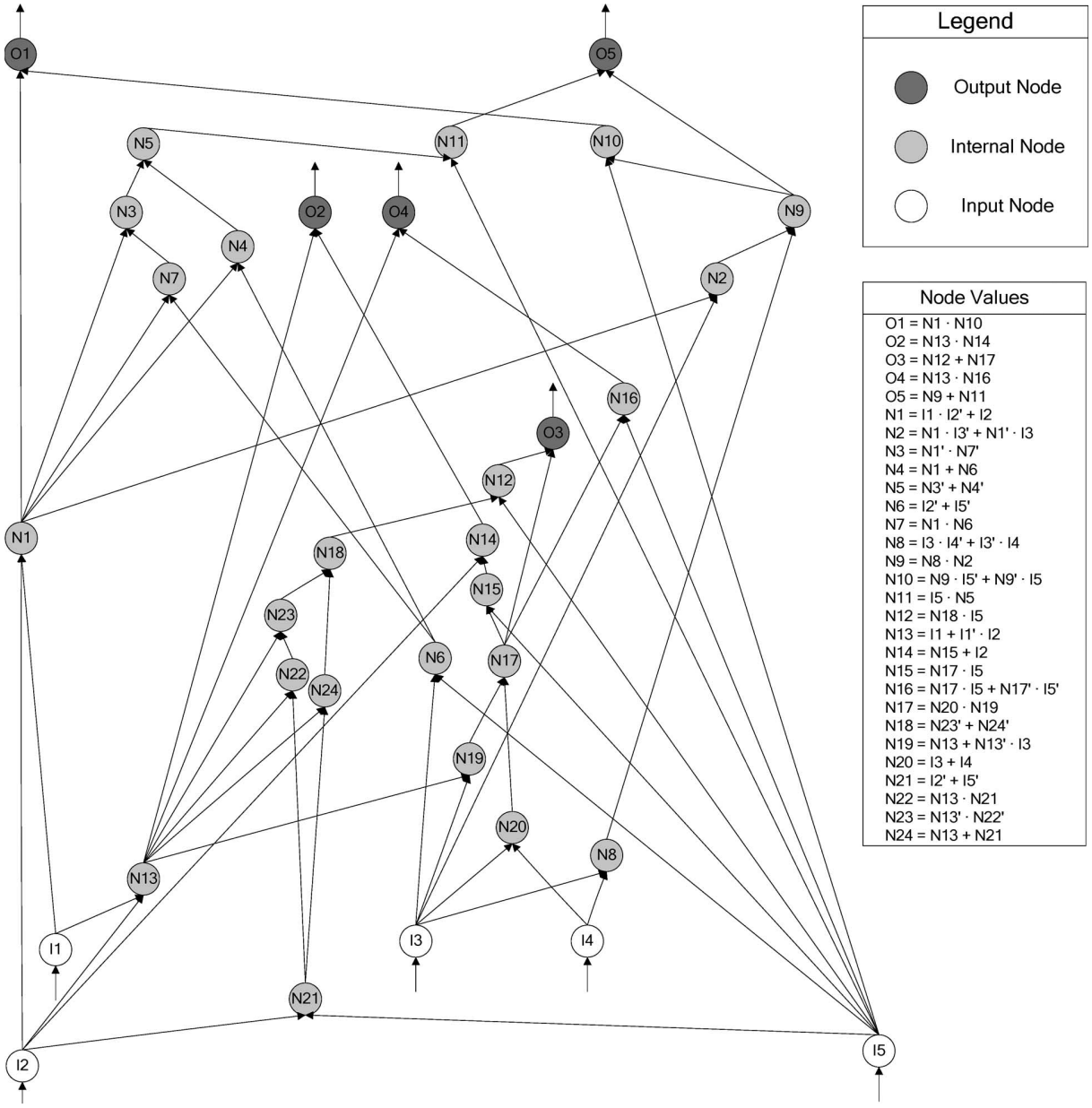



Figure 3.2: Representation of C5 circuit from (CONG; MINKOVICH, 2007).

3.3 with $(n-1)$ levels. A circuit with $(n-1)$ levels, with each node representing a binary operator, can have at most $2^{(n-1)}$ leaves, contradicting the assumption.

Even though the proposed method guarantees trees with exact logic depth, it may lead to trivial functions, which could be treated as a particular case by the logic synthesis computing engines. For instance, if the generated tree has only AND/OR operators, it generates a unate function (ALON; BOPPANA, 1987). Therefore, the tree would be easily synthesized by dedicated methods, such as the one presented in (THORP; YEE; SECHEN, 1999). In order to avoid these functions, after assigning the nodes randomly, the authors propose to ensure the presence of at least one binate operator (exclusive-OR

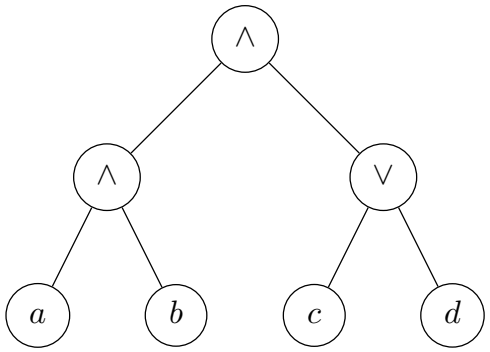



Figure 3.3: Depth optimal realization of $f = abc + abd$ (AMARÚ et al., 2017).

or exclusive-NOR). The binate operator can be randomly assigned and is enough to end up the possibility of the function be handled as a special case by methods dedicated to unate functions. Figure 3.4 shows a possible assignment of the exclusive-OR (XOR) operator in order to break the unateness feature of the trivial tree presented in Figure 3.3.

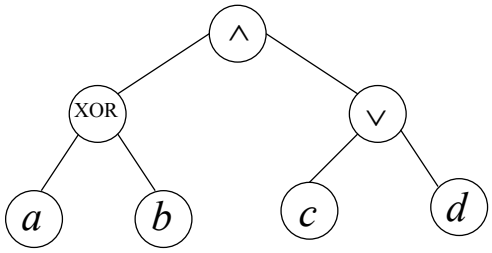



Figure 3.4: Breaking unateness of tree presented in Figure 3.3.

Still, the resulting tree is disjoint support for any node (BERTACCO; DAMIANI, 1997). Therefore, a further improvement is proposed to avoid the generation of trees representing disjoint support functions, which can also be efficiently handled by special algorithms implementing disjoint support decomposition (DSD) (MISHCHENKO; BRAYTON, 2007). To avoid such functions, the authors generate two random trees with the same number of input variables, but with different functionality. Since the trees are generated randomly, it is unlikely that both represent the same function (AMARÚ et al., 2017). Then, both trees have their inputs shared, and their roots merged with a binate operand, leading to a unique output, as presented in Figure 3.5. The addition of this new node results in a tree with $(n + 1)$ levels. In the next step, a technique based on Boolean satisfiability is adopted to guarantee that the final tree depends on all its $2^n$ variables (SOEKEN et al., 2016a). Guaranteeing that the function depends on all its variable leads to a binate tree, where at least one node does not accept disjoint support decomposition techniques (MISHCHENKO; BRAYTON, 2007).

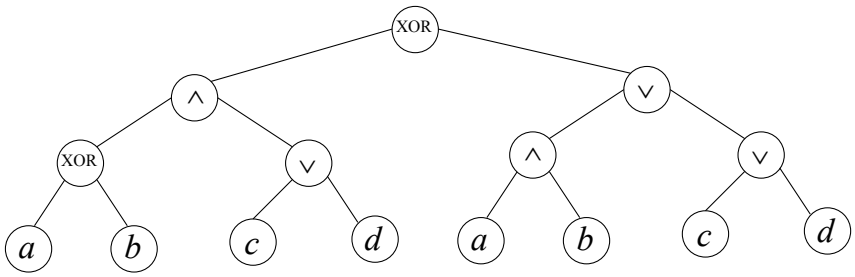



Figure 3.5: Breaking disjoint support of tree presented in Figure 3.4.

At the end of the process, the method proposed in (AMARÚ et al., 2017) it is able to generate a tree with exact logic depth. The final tree has $2^n$ inputs, $(n+1)$ levels because the operator added to join both roots and $(2^{(n+1)}-1)$ nodes. Although the depth optimality is guaranteed at the binary tree, it is not guaranteed when this tree is transformed in an AIG. Thus, the authors run the *"rewrite"* command available on the ABC tool to recover the depth optimality in the AIG level (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

From the AIG with optimal logic depth, it is generated binary decision diagram (BDD) representing the target circuit (BRYANT, 1986). The BDD of a given circuit has as many levels as inputs. Hence, the circuit with an optimal logic depth of $(n+1)$ would be collapsed into $2^n$ levels. This resulting circuit serves as a sub-optimal starting point to feed the synthesis tools. At the end, the work presents an exponential gap between results found by tools and the optimal solution. Results are presented for AIGs with up to 600,000 nodes.

Besides having exact benchmarks for a given metric (*i.e.,* circuit area or logic depth), it is also important to have available those circuits with different sizes. Therefore, it becomes easier to assess why a method does not find the expected solution, and how does the method scale as circuits become huge. This work presents a novel method to automatically generate exact benchmark circuits.

# 4 PROPOSED APPROACH

In this work, we propose a new method for designing exact multi-level logic circuits in both circuit size (number of nodes) and logic depth. The proposed method has a proven known optimal solution at the AIG level and after the technology mapping process. That is possible because our method relies on the principle of reversible logic, and the final circuit implements an identity function $F(x) = x$. So that, we implement a transparent logic, which could be simplified to wires by the logic synthesis tool.

To do so, the method proposes the design of a logic block comprising two stages: the first stage performs a reversible function $F$, whereas the second one implements another reversible function corresponding to the inverse function $F^{-1}$ of the first stage one. As a result, by combining both stages, the logic behavior of the second stage output is equivalent one-by-one to the input logic value of the first stage. Therefore, the resulting block implements an identity function corresponding to $F(x) = x$. The resulting identity logic block (ILB) diagram is shown in Figure 4.1.

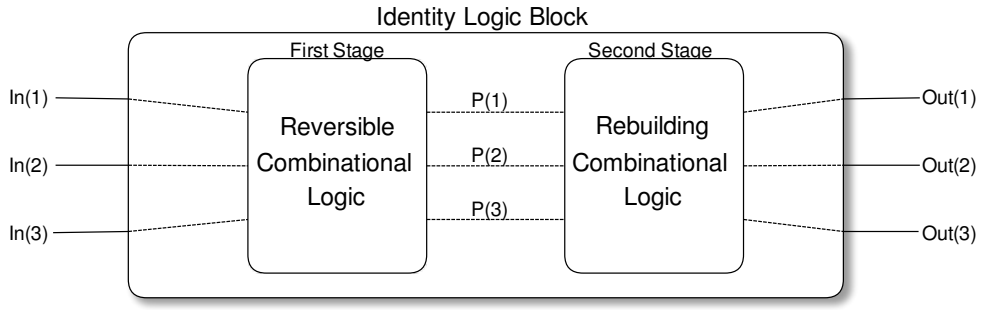



Figure 4.1: Identity logic block (ILB).

There are different possibilities for generating the first stage of ILBs. At first, it is possible to derive such a stage from any standard (irreversible) logic circuit. To do so, one alternative is to take a real design (usually implementing an irreversible function) and then embedding it into a reversible function. It can be done by applying known embedding methods, such as the ones presented in (SOEKEN et al., 2016b) and in (ZULEHNER; WILLE, 2017).

In this work, we address the issue of deriving exact blocks from real designs, which are often used as benchmark circuits, by exploring open-source embedding tools. We derive the ILB first stage from three different sized circuits, ranging from a very simple design to more complex ones. The ILB generation from irreversible functions is discussed in Section 4.1.

Another possibility is to generate a synthetic logic block and guarantee its reversibility by construction. Such an alternative is addressed in Section 4.3, where an algorithm to generate exact synthetic circuits is presented. Generating synthetic circuits through the proposed algorithm is useful to keep control on the number of inputs and outputs nodes (I/O) of the circuit, so that it is easier to figure out when the logic synthesis methods stop to finding the exact solution. Work in this frontier between finding or not the exact solution may lead to new methods.

Besides the possibilities to generate an exact multi-level logic circuit exploiting reversible logic, whether by construction or by embedding, the proposed approach also unlocks the possibility to embedding any custom logic in the ILB block, as discussed in Chapter 5. The custom logic can be exact and be built by performing exact synthesis methods over a given function or by incorporating previously proposed exact synthesized circuits. Embedding a custom block with a known solution can stress different features on the target logic synthesis algorithm. It is also possible to take a custom logic with the previously known best synthesis result (which may not be the optimum solution) and incorporate it into the ILB. This approach is also discussed in Chapter 5. It enables the possibility to assess how the redundant logic disturbs the synthesis tool, representing a kind of noise in the logic synthesis process.

A summary of methods for exact multi-level logic circuit generation is presented in 4.1. The BBT name refers to the balanced binary tree method proposed in (AMARÚ et al., 2017). It is possible to note that the method proposed in this work is the only one to accomplish the main features for exact benchmarks. The interrogation mark about deriving LEKO from real designs is because the authors argue that it is possible and all one need to do it is to extract a sub-graph from the circuit AIG (CONG; MINKOVICH, 2007). However, it is nor described neither clear in the reference how this graph will respect the features required by LEKO, *i.e.*, each output depending on all inputs.

Table 4.1: Approaches for exact multi-level logic circuit generation.

| Name | Year | Exact Area | Exact Logic Depth | Synthetic | From Real Design |
|---|---|---|---|---|---|
| LEKO/LEKU | 2007 | Yes | No | Yes | ? |
| BBT Based | 2017 | Yes | Yes | Yes | No |
| This Work | 2018 | Yes | Yes | Yes | Yes |

## 4.1 Deriving ILB from Real Designs

To derive circuits from real designs, the proposed method relies on reversible logic, which has had a growth in research in the last decade. Such renewed interest in this topic has been mainly motivated by reversible logic applications in low power designs and beyond complementary metal-oxide-semiconductor (CMOS) based integrated circuits.

Even though there are well-established techniques to reduce power, there will always be some energy dissipated per bit of information lost, regardless of the target technology. This observation was firstly introduced by Rolph Landauer, in (LANDAUER, 1961), and experimentally validated by Bérut, in (BÉRUT et al., 2012). Landauer stated that using traditional (irreversible) logic leads to a minimal of heat generated for each bit of information lost, which is at least $(k.T.ln(2))$, where $k$ is the Boltzmann constant and $T$ is the room temperature. Later, in (BENNETT, 1973), Bennet demonstrated that zero energy dissipation is only possible in a circuit where there is no information lost, *i.e.*, a reversible circuit.

Besides the application in low power, reversible logic also finds use in emerging technologies, such as quantum computing (NIELSEN; CHUANG, 2002), DNA computing (KLEIN; LEETE; RUBIN, 1999) and optical computing (KNILL; LAFLAMME; MILBURN, 2001). As discussed in (NIELSEN; CHUANG, 2002), quantum computing is of particular interest because it was shown that quantum algorithms could solve problems in polynomial time, whereas for convention algorithms only exponential methods exist. Furthermore, all quantum operations are reversible. Therefore, synthesis of reversible functions has become an deeply studied topic.

In this work, we propose a novel application for reversible logic, *i.e.,* generating exact multi-level logic benchmarks. In order to achieve this goal, we explore previously proposed methods of embedding, which were developed targeting reversible logic synthesis. For a further discussion on reversible logic synthesis, please refer to (WILLE; DRECHSLER, 2010)(SAEEDI; MARKOV, 2013). With the embedding methods, we achieve the goal of generating exact benchmarks from real functions.

The possibility of generating exact circuits from real designs is particularly interesting since while creating these circuits by synthetic construction, as in (CONG; MINKOVICH, 2007) and in (AMARÚ et al., 2017), it is not possible to guarantee that it is composed by structures implementing logic commonly found in real designs. In other words, a random construction may lead to circuits computing meaningless functions. In

this sense, we can end up tuning our algorithm to optimally synthesize functions that are not likely to appear in real applications. Therefore, embedding methods of irreversible functions unlock the possibility of deriving exact circuits from designs with some practical and useful functionality. Therefore, programmers can tune their tools concerning real designs.

To use real designs as seed to generate exact benchmarks, first of all its functionality must be represented as a reversible function. To convert an irreversible function into a reversible one, one does what is known as *embedding*. The embedding process for an irreversible function consists of adding $k$ inputs and $g$ outputs such that, at the end, the number of inputs and outputs must match. Furthermore, the newly added outputs are *don't cares*, and must be assigned such that each input pattern must map a unique output, *i.e.,* a bijection. This process is quite complex, being coNP-hard (SOEKEN et al., 2016b)), and, in the worst case, all inputs/outputs relationship must be addressed. Usually, the extra inputs are constants, known as *ancilla*, while the additional outputs are identified as *garbage outputs*.

The first embedding methods were based on truth tables. With such representation, the minimum overhead regarding extra inputs and outputs is easily calculated. For instance, let's consider the full adder function described in Table 4.2. Since the output patterns $(sum = 1, cout = 0)$ and $(sum = 0, cout = 1)$ appear three times each, the full adder function does not implement a bijection, being irreversible. Furthermore, the number of inputs and outputs are different.

Table 4.2: Irreversible full adder truth table.

| *x0* | *x1* | *x2* | *sum* | *cout* |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Therefore, let $\mu$ denotes the times the most frequent output pattern occurs (in the full adder case $\mu = 3$), then it is necessary to add the following number of garbage outputs

to distinguish all the occurrences of the pattern:

$$l = \lceil \log_2 \mu \rceil \tag{4.1}$$

This formula gives us both the upper and lower bound, being that an optimal bound (SOEKEN et al., 2016b). Thus, if we add $l$ garbage outputs in the embedding process, the embedded is said optimal because it has the minimum overhead regarding additional inputs and outputs. Notice that this optimum does not relate to the optimal implementation of the circuit. An optimal implementation depends on the don't cares assignment after adding the constant inputs and the garbage outputs (WILLE; DRECHSLER, 2010).

Table 4.3 presents a straightforward embedding of the full adder into a reversible function. Notice that there exist no distinct input pattern mapping the same output. Since the most frequent pattern appears three times, by using the equation 4.1, $\mu = 2$. It means that two garbage outputs must be added and, consequently, one constant input will be added to keep $n = m$. In Table 4.3, the constant input is denoted by $k$ while the garbage outputs are denoted by *g1* and *g2*.

Table 4.3: Reversible full adder truth table.

| *k* | *x0* | *x1* | *x2* | *sum* | *cout* | *g1* | *g2* |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

However, even though truth table based methods are straightforward and may lead to a minimal overhead, they do not scale for large functions since the truth table size increases exponentially with respect to the number of input variables. Therefore, new

embedding methods have been proposed targeting scalability. In (SOEKEN et al., 2016b), the authors proposed two ways, the first is exact and based on cube representation, and the second is heuristic and based on BDD (AKERS, 1978). More recently, a new method based on matrix representation was proposed in (ZULEHNER; WILLE, 2017).

In this work, we have adopted the embedding method based on BDD, presented in (SOEKEN et al., 2016b). That is because the method is publicly available on the RevKit tool (SOEKEN et al., 2012), unlike the one presented in (ZULEHNER; WILLE, 2017), and scales better than the exact cube based approach.

The heuristic BDD based embedding relies on a BDD structure that may be created by any algorithm. The designs used in this work were read either in Verilog or PLA format and then converted to BDD utilizing a set of commands available on the RevKit tool. This embedding algorithm is adapted from the initial idea of embedding proposed by Bennett, in (BENNETT, 1973).

Bennett has proven the upper bounds in the number of additional outputs as follows:

**Theorem 1.** *For a function $f \in B_{n,m}$, at most n additional outputs are required to embed $f$.*

*Proof.* The number of additional outputs is maximized if, for a given output $y \in B^m$, we have $f(x) = y$ for all $x \in B^n$. Therefore, there are $2^n$ repetitions and the number of additional outputs, from equation 4.1, is given by $l = \lceil \log_2 2^n \rceil = \text{n}$. □

With this upper bound, Bennett has applied an explicit embedding described by the following *theorem*:

**Theorem 2.** *A function $f \in B_{n,m}$ is embedded by the function $g \in B_{m+n,m+n}$, where*

$$g(k_0, ..., k_{m-1}, x_0, ..., x_{n-1}) = (y_0, ..., y_{m-1}, g_0, ..., g_{n-1}). \tag{4.2}$$

*with*

$$y_i(k_0, ..., k_{m-1}, x_0, ..., x_{n-1}) = k_i \oplus f_i(x_i, ..., x_{n-1}) \tag{4.3}$$

*and*

$$g_i(k_0, ..., k_{m-1}, x_0, ..., x_{n-1}) = x_i \tag{4.4}$$

To exemplify it, lets take as example the 2-input AND Boolean operator, presented in Table 4.4.

| *x0* | *x1* | *y0* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Table 4.4: 2-input AND truth table.

From the Theorem 2, it is known that the resulting function has $(n+m)$ inputs and outputs, in this example $(n + m = 3)$. The resulting embedded function $g(k_0, x_0, x_1) = (y_0, g_0, g_1)$ is presented in Table 4.5. The value of the function output $y_0$ is given by equation 4.3, *i.e.*, if the constant input $k_i = 0$ then the output receives its own value $(x_0 \wedge x_1)$, else if $k_i = 1$ then $y_0$ receives its inverse. In turn, the *garbage outputs* are given by equation 4.4, that is $g_0 = x_0$ and $g_1 = x_1$.

To unlock this embedding procedure in large functions, in (SOEKEN et al., 2016b), the authors adapt the Bennett's embedding to run over BDD. After the embedding, the circuit is stored in a data structure called as RCBDD (SOEKEN et al., 2016b). From the RCBDD, it is possible to write the reversible function specification into an output file described in the PLA format.

| *k0* | *x0* | *x1* | *y0* | *g0* | *g1* |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 |

Table 4.5: Bennett's embedding of 2-input AND.

## 4.2 Processing Embedded Output

This section presents an example of a full adder embedded with the RevKit tool, as well as the resulting function specification. We show that the function specification given by the tool output is not suitable to generate the ILB so that an algorithm to enable the ILB generation is presented.

The full adder presented in Table 4.2 was described in PLA format and used as input to the RevKit tool. The PLA was read in RevKit and converted to a BDD by using commands provided by the tool. In the sequence, the BDD based embedding was run and the resulting reversible circuit, described in an RCBDD, was wrote back in PLA format. However, instead of having a complete description in the output PLA, such as the one in Table 4.3, what we get from the method is the function specification for the constant input $k$ equals to 0. Therefore, the truth table provided by the method has $2^{n-1}$ rows. For the full adder case, the returned truth table looks like the one presented in Table 4.6.

Table 4.6: Reversible truth table returned by RevKit tool for full adder.

| *k0* | *x0* | *x1* | *x2* | *sum* | *cout* | *g0* | *g1* |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |

Since the function specification is incompletely specified, that is not possible to directly derive an ILB from that. Let's consider, for instance, the embedding function g($k$,$x_0$,$x_0$,$x_1$,$x_2$) = (*sum*,*cout*,$g_0$,$g_1$) when $g(0,0,0,1) = (1,0,0,0)$. Trying to inverse this specification leads to the function $g(1,0,0,0)$. However, this input pattern is not specified in the function description (it is a *don't care*). Therefore, it is not possible to guarantee the bijection feature for the first stage of the ILB if it is obtained directly from the RevKit embedding method.

To quickly completely specify the function, we propose an algorithm composed of three main steps, as follows:

1. Read the inputs and outputs from the PLA output given by RevKit tool into both a vector and a hash table, serving as a look-up table.

2. Iterate over each input pattern and search for it on the outputs hash table. If the hash does not contain the element, the pattern is added in both, the output vector and the output look-up table. The same is done from the outputs to the inputs.
3. Generating the missing inputs and outputs relationship, and adding them to both the hash table and the vector.

The hash table serves as a look-up table to store and search patterns that were already considered. However, we also use a vector since the hash table does not keep the order of the processed elements. At the end, the patterns stored in the input and output vectors are added to the original PLA file, resulting in a completely specified function which keeps its initial output values. Through this method, a truth table of a 4-input arithmetic logic unit with 19 I/O (after embedding) was completely specified in less than 4 seconds.

After the PLA processing by the described algorithm, the resulting file is read into the ABC tool (Berkeley Logic Synthesis and Verification Group, 2018). ABC is used to create a multi-level description of the resulting circuit, which is written in a BLIF file. This resulting file implements the reversible function $F$ of the ILB first stage, as presented in Figure 4.1.

To create the second block implementing the inverse function, $F^{-1}$, all we need to do is to swap the input and output patterns from the first block PLA file, *i.e.,* specify a circuit to undo the first block logic. This inverse block is also described in PLA format, and ABC is again used to provide the BLIF description.

Both BLIF files are then connected through a *script* and the final ILB description is done. To combine both BLIFs, all one needs to ensure is that the first stage inputs are the ILB primary inputs, the first stage outputs are internal nodes connected to internal nodes representing the second stage inputs, and the second stage outputs are the ILB primary outputs.

Even though this method relies on real circuits to generate benchmarks, it tends to be restricted to small designs. That is due to the complexity of the process in turning irreversible function into reversible one, which does not scale for large designs. Therefore, in order to have unlock the generation of complex circuits with known solution, the next section discusses the proposed method to generate synthetic benchmarks. The exact solution is guaranteed by construction.

## 4.3 Generating ILB by Construction

This section discusses how to generate synthetic circuits with a known exact solution. Our interest in creating synthetic circuits is to keep control on the benchmark size, with the goal of discovering for which sizes open-source and commercial tools can find the exact solution, as well as verifying when these tools start to have problems while optimizing the circuit.

To derive the synthetic ILB, we need to generate each of its stages apart. Therefore, the first stage computes a reversible function $F$. From the primary stage function $F$, we need to design the second stage to perform $F^{-1}$.

*ILB First Stage*: The first stage corresponds to a circuit implementing a randomly generated reversible multi-output Boolean function $f$, with $n$ inputs and $m$ outputs, being $m = n$. To describe a completely specified function, we have adopted a truth table. Each row of the truth table output is represented as an integer, and our truth table can be represented through a vector of $2^m$ positions. To guarantee the reversibility of the generated function, we start by enumerating unique positive integers from 0 to $(2^m - 1)$ and storing each of these integers into a different index of the vector. Afterwards, a random shuffle procedure from C++ standard library is applied over this vector, leading to our random function. From that, we can construct our truth table, where the output bits of the $i_{th}$ row are defined by the binary expansion of the $i_{th}$ integer of the vector. The input bits are given by the binary expansion of the $i_{th}$ index.

From the presented construction algorithm, it is easy to note that the generated function is a bijection. Since the integers are uniquely enumerated, there are not repeated patterns at the output, keeping a one-to-one mapping from inputs to outputs.

At the end, the generated truth table describing the ILB first stage is written in PLA format. This file serves as input to the ABC tool in order to obtain the BLIF description of the first block, which is connected to the second block.

Figure 4.2 presents the flow to generate the first block of an arbitrary 3-input function. The numbers above the boxes represent the index of each element stored on the vector. In the generated truth table, $i(1), i(2)$ and $i(3)$ are equivalent to the PIs of the ILB $In(1), In(2)$ and $I(3)$ shown in Figure 4.1. P$(x)$, with $1 \leq x \leq m$, in the generated truth table represents the outputs of the first block, which are intermediate outputs of the ILB, as shown in Figure 4.1.

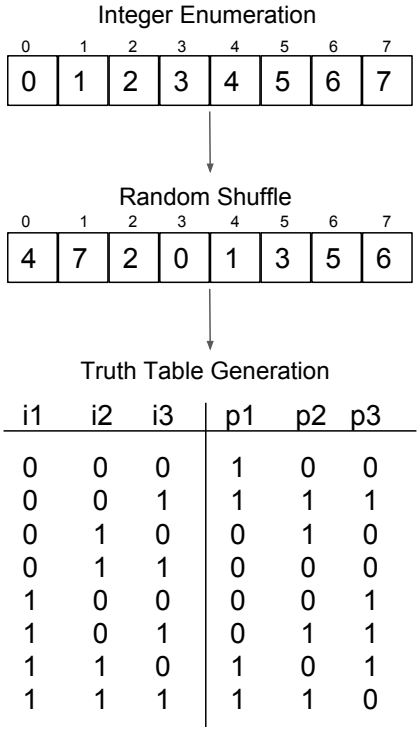



Figure 4.2: Flow for generating the ILB first stage.

*ILB Second Stage*: To generate the second stage, the intermediate nodes of the ILB, *i.e.*, the outputs of the first stage, serve as the inputs of the second stage. Hence, this second stage is designed to rebuild the primary inputs of the first stage in its output, so that the proposed ILB computes the identity function, *i.e.*, $In = Out$. The second stage is easily obtained from the first stage structure. Here, the input bits of the $i_{th}$ row are defined by the binary expansion of the $i_{th}$ integer in the output vector. On the other hand, the outputs bits of the truth table $i_{th}$ row is defined by the binary expansion of the $i_{th}$ vector index.

The second stage is written as PLA file by the proposed method and serves also as input of the ABC tool to generate its BLIF description. In the sequence, we connect the BLIF description of each stage, resulting in a BLIF describing the whole ILB, as presented in Figure 4.1. As the last step, this file is read in ABC tool and we obtain its AIG representation by running the ABC command *strash*. The resulting AIG has its known exact solution in both circuit logic depth and area (zero), and serves as the start point to assess the logic synthesis algorithms.

Table 4.7 presents the truth table of the ILB generated from the flow depicted in Figure 4.2, with the corresponding intermediate nodes. In(x) and Out(x) denote the PIs and POs of the ILB, respectively, with $1 \leq x \leq (n,m)$. From this truth table, it is possible to note that the resulting block implements an identity function and could be reduced to

Table 4.7: Final ILB of an arbitrary three inputs function.

| In(1) | In(2) | In(3) | P(1) | P(2) | P(3) | Out(1) | Out(2) | Out(3) |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

wires.

The trends in the size of the synthetic circuits are presented in Table 4.8. The first column represents the number of inputs and outputs of each circuit, followed by the number of AIG nodes and circuit logic depth for the original ILB. Even though the presented approach can generate circuits with more than 21 I/O, the ABC tool takes more than one day to produce the corresponding AIG, so we have considered time out. It is possible to note that the AIG size and logic depth from the created synthetic circuits are exponential with respect to the number of the circuit I/O. Also, it is shown that the proposed approach can generate huge circuits in the number of AIG nodes, which tends to impose difficulties to the state-of-art synthesis algorithms.

Table 4.8: Number of nodes and logic depth for synthetic circuits.

| I/O | # nodes | depth |
|---|---|---|
| 2 | 6 | 3 |
| 3 | 25 | 8 |
| 4 | 83 | 12 |
| 5 | 206 | 17 |
| 6 | 489 | 22 |
| 7 | 1,115 | 28 |
| 8 | 2,496 | 32 |
| 9 | 5,616 | 37 |
| 10 | 12,170 | 44 |
| 11 | 25,714 | 49 |
| 12 | 55,176 | 56 |
| 13 | 116,879 | 63 |
| 14 | 244,561 | 69 |
| 15 | 515,165 | 78 |
| 16 | 1,068,922 | 88 |
| 17 | 2,221,295 | 99 |
| 18 | 4,564,853 | 114 |
| 19 | 9,395,112 | 136 |
| 20 | 19,626,463 | 159 |
| 21 | 39,978,858 | 205 |

# 5 EXACT BENCHMARKS SYNTHESIS RESULTS AND DISCUSSION

This chapter presents and discusses the experimental results for the synthesis of ILBs based on both real designs and synthetically generated ones. Furthermore, the use of ILBs in more complex scenarios is also evaluated. Experimental results are presented for the state-of-art open source tools as well as commercial tools, taking into account standard cell ASIC design and LUT-based FPGA synthesis.

## 5.1 Synthesis Results for ILBs Based on Real Designs

The first experiments of ILB synthesis based on *real designs* were carried out using the open-source tools ABC and CirKit (Mathias Soeken, 2018). ABC is widely used in academia for logic synthesis processes whereas CirKit has been chosen because it supports synthesis over MIGs, which is a promising data structure.

Considering that the ILB implements an identity function that could be replaced by wires, *i.e.,* zero logic gates, we have chosen for both tools commands to reduce the AIG and MIG sizes. Reaching the optimal solution in the number of nodes will also lead to the optimal solution in circuit logic depth. To do so, we have opted per area optimization commands that are not constrained by logic depth preservation. It means that the command always chooses to reduce the circuit area, even though it may increase the logic depth. It is worth to notice that the area oriented commands from ABC are based on AIG rewriting (MISHCHENKO; CHATTERJEE; BRAYTON, 2006), which is a standard for both academia and industry (AMARÚ et al., 2017). On the CirKit side, the chosen command was the *mig_rewrite*, enabling the flag that sets the area as the cost metric for optimization. To run the *mig_rewrite*, the AIG is first converted to a MIG using CirKit commands. The rewrite command is then run and, at the end, the MIG network is converted back to the AIG one for a fair comparison. We stop running each command when the AIG (MIG) size is no longer reduced during five successive iterations.

Table 5.1 presents the synthesis results. The *Depth* column presents the AIG logic depth after embedding and before any optimization. In the sequence, the number of I/O for each circuit is presented for both the original design and the embedding result. Finally, the column *nodes* presents the original number of nodes after embedding and the resulting number of nodes after running a given command. Time-out means that the command has not finished its execution after one day running, being then aborted.

The first tested circuit was the 4-input arithmetic logic unit (*alu4*). This circuit was chosen because it has arithmetic structures, which are often found in real applications. Since none of the commands could find the expected solution for this circuit, a smaller circuit should be taken in place to assess if any algorithm can find the exact solution.

Therefore, the *z4ml* circuit, from the LGSynth'93 was chosen (MCELVAIN, 1993). From results, we can note that none of the rewriting based commands could find the exact solution. In fact, the *dc2* command, which has had the best result among the rewriting based commands, provides results still very far from the expected solution. Moreover, we have run the *dsd* command which applies the disjoint-support decomposition. For this command, it is expected that it can reach the minimal solution for the proposed circuits since its underlying data structure for Boolean decomposition is a BDD (BERTACCO; DAMIANI, 1997). Therefore, when considering the proposed case studies, the exact solution with zero nodes can be found as soon as the BDD construction is finished. Even though the *dsd* command is able to run and found the solution for small circuits, we are still interested in assessing the rewriting based commands. The objective while evaluating these commands is to figure out for which size of circuits it was possible to reach the exact solution.

Hence, an even smaller circuit was taken from the ISCAS'85 benchmark (BRGLEZ; FUJIWARA, 1985), the *c17*. This circuit is initially composed of six 2-input NAND logic gates and it is one of the simplest circuits for benchmarking. Still, even for this simple circuit, with a few hundreds of nodes, the commands were not able to find the exact solution. In fact, all commands are much closer to the original AIG size than the optimal solution. Thus, it is possible to conclude that the current state-of-art algorithms available on open-source tools may be improved.

Notice that there is a considerable gap between the size of *z4ml* and *alu4* circuits. Therefore, it would be interesting to identify a thinner frontier where the synthesis commands are able to find or not the optimal solution.

Table 5.1: Results for ILBs based on real designs in open-source tools.

| | | I/O | | # nodes | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Circuit | Depth | Original | Embedded | Original | dc2 | mig_rewrite | drw | compress2rs | dsd |
| alu4 | 86 | 14/8 | 19 | 443,227 | 274,089 | 441,138 | 350,114 | 317,996 | time-out |
| z4ml | 34 | 7/4 | 8 | 1,898 | 1,224 | 1,886 | 1,529 | 1,320 | 0 |
| c17 | 22 | 5/2 | 6 | 390 | 251 | 386 | 314 | 248 | 0 |

The benchmark circuits mentioned above were also synthesized by applying a standard commercial tool. To perform this experiment, the ILBs were converted to Verilog

format, from the AIG representation, by using the ABC tool. The Verilog file was then used as input for the synthesis tool, as well as a simple constraint file.

Table 5.2 presents the solutions for the commercial tool. The first column presents the design name and the second column presents the AIG depth. The number of I/O for the original design and after embedding is presented in the third column. The original AIG number of nodes is presented in the column *nodes*, while the commercial synthesis results are presented in terms of the number of logic gates in the last column.

Table 5.2: Results for ILBs based on real designs in commercial tool.

| | | I/O | | #nodes | #gates | |
|---|---|---|---|---|---|---|
| Circuit | depth | original | embedded | | generic | mapped |
| alu4 | 86 | 14/8 | 19 | 443,227 | 172,632 | 188,485 |
| z4ml | 34 | 7/4 | 8 | 1,898 | 3,390 | 0 |
| c17 | 22 | 5/2 | 6 | 390 | 714 | 0 |

Notice that for the *z4ml* and *c17* circuits the generic results have more gates than the original number of AIG nodes. That is because the number of gates also considers inverters, while in the AIG the inverters are a parameter on the graph edges. For the *alu4* circuit, it is possible to see that there are more gates in the mapped circuit than in the generic one. That is because there are more inverters and buffers in the mapped circuit. If we consider only logic gate instances, the generic synthesis has fewer gates than the original AIG, and the mapped synthesis has lesser gates than the generic one.

The synthesis results are presented for two steps on the logic synthesis flow. The generic column stands for results provided by the tool while optimizing the circuit before proceeding to the technology mapping task. In turn, the mapped presents the number of logic gates after mapping the circuits based on a given standard cell library.

In respect to the generic synthesis results, it is expected that the minimum solution could be found at this phase, *i.e.*, through the application of technology independent optimizations. However, as the commercial tools are black boxes, it is quite difficult to understand and predict their behavior.

Therefore, we have proceeded in the logic synthesis flow by running the technology mapping after the generic synthesis. As discussed in Chapter 2, mappers can be classified as structural or functional. The experimental results presented in Table 5.2 show that the exact solution was just found after technology mapping. Hence, we have a clue that

more powerful optimizations are performed in the commercial tool during the technology mapping, which may indicate that the tool implements a functional mapper.

Notice that there is a considerable gap in the size of the circuits that the tool can or can not find the exact solution. It also motivates us to generate synthetic circuits keeping control on the size, so that we can check when the commercial tool starts having difficulties.

## 5.2 Synthesis Results for Synthetic ILBs

We propose a progressive approach to validate the proposed method. At first, we have incrementally generated circuits starting from $n = m = 2$ up to $n = m = 21$. Then, we have executed the same set of commands we have used to synthesize the real designs. The optimization commands are run over the generated AIGs, targeting area reduction without being constrained by the circuit logic depth. In the same way that in the experiments describe before, we stop running each command when the AIG size is no longer reduced during five successive iterations. For the *mig_rewrite* command, the same procedure was adopted to run MIG optimization.

The results for the synthetic ILBs synthesis are presented in Table 5.3. The first three columns show the AIG I/O, size and depth, respectively. In the command columns, $y$ indicates that the command has found the exact solution. Otherwise, it is presented the final number of AIG nodes after stop running the command. We are showing results up to $n = m = 15$ because most of the commands have already stopped finding the optimum number of nodes for $n = m = 5$. For the *dsd* command, it starts timing out (the command have not finished after one day executing) for circuits with 14 I/O.

The results presented in Table 5.3 can be justified by the intrinsic characteristics of the applied synthesis algorithms. Most of these algorithms are based on tables of precomputed structures, which are used for replacing subgraphs defined by 4-input cuts (MISHCHENKO; CHATTERJEE; BRAYTON, 2006). Notice that the exact solutions are often found for the case studies with up to four inputs. In these cases, even by using 4-input cuts, the algorithms have a global view of the related AIG due to their reduced number of inputs and AND nodes. However, the local nature of such optimizations is strongly associated to the problem of escaping from local minima for larger AIGs.

Table 5.3: Results for size (area) oriented commands.

| I/O | size | depth | *dc2* | *mig_rewrite* | *drw* | *compress2rs* | *dsd* |
|---|---|---|---|---|---|---|---|
| 2 | 6 | 4 | 0 | 6 | 0 | 0 | 0 |
| 3 | 25 | 8 | 0 | 24 | 0 | 0 | 0 |
| 4 | 83 | 12 | 0 | 78 | 0 | 0 | 0 |
| 5 | 206 | 17 | 151 | 203 | 173 | 148 | 0 |
| 6 | 489 | 22 | 345 | 483 | 396 | 346 | 0 |
| 7 | 1,115 | 28 | 808 | 1,107 | 924 | 837 | 0 |
| 8 | 2,496 | 32 | 1,818 | 2,480 | 2,090 | 1,889 | 0 |
| 9 | 5,616 | 37 | 4,021 | 5,585 | 4,652 | 4,262 | 0 |
| 10 | 12,170 | 44 | 8,574 | 8,323 | 10,108 | 9,523 | 0 |
| 11 | 25,714 | 49 | 18,017 | 25,623 | 21,403 | 20,136 | 0 |
| 12 | 55,176 | 56 | 37,970 | 54,936 | 45,734 | 42,629 | 0 |
| 13 | 116,879 | 63 | 79,490 | 116,424 | 96,663 | 91,009 | 0 |
| 14 | 244,561 | 69 | 163,588 | 243,480 | 202,097 | 189,627 | time-out |
| 15 | 515,165 | 78 | 336,760 | 512,754 | 422,805 | 397,834 | time-out |

In order to verify whether increasing the number of AND nodes in the circuits with up to 4 inputs would change the algorithm behavior, we have cascaded ILBs with the same amount of I/O in a chain configuration. By doing that and re-running the commands, the results are quite similar. It can be explained by the fact that all blocks have the same number of I/O. Therefore, the command computes $k$-cuts to deal with the first stage and simplifies it to a wire (no logic included). In that sense, while dealing with the next ILB, the previous one has already been consumed, and the $k$-cuts are computed at the PIs of the next ILB. By the end, it becomes equivalent to synthesize each ILB independently.

Regarding the *dsd* command, as previously explained, it is expected that this method can reach the exact solution since its underlying data structure for Boolean decomposition is BDD (BERTACCO; DAMIANI, 1997).

Finally, in order to evaluate whether the randomness during the generation of the proposed benchmarks brings a bias to the experiments, we have generated and assessed ten more circuits with four and five I/O nodes. This experiment has not presented any changes in the results, *i.e.*, on the frontiers between exact and non-exact solutions. Therefore, these results are not explicitly shown herein. However, with such an experiment, we could evaluate that the randomness of our approach is not crucial when looking for the frontiers of finding or not exact solutions.

Notice that, similarly to the analysis presented in (AMARÚ et al., 2017), the primary goal of our experiments is not to compare the available synthesis methods. Instead, we are concerned to figure out the frontiers where algorithms stop to find out exact solutions. Determining such a boundary is interesting because it enables to select small case

studies where the exact solution is not found, contributing to the effort toward possible improvement in logic synthesis algorithms.

The same set of synthetic circuits used in the open-source tools were also synthesized by a commercial tool. For the same reason, as discussed before, the generic synthesis result has more logic gate instances than the original number of AIG nodes. That is, in AIG representation, inverters are represented through a parameter on the graph edges and are not counted. On the other hand, inverters and buffers are counted on the report provided by the commercial tool.

As in the results presented for benchmarks based on real circuits, the generic synthesis was not able to find the exact solution, even for the smallest circuit, with two I/O. Again, it was expected that this step could reduce the circuits to its optimal implementation through technology independent optimization.

Proceeding to the technology mapping, the commercial tool was able to optimally synthesize the circuits with up to 25,000 AIG nodes. This result gives a better notion for which circuit size the commercial tool can find the optimal solution for circuits implementing an identity function, and when it starts to have problems. Moreover, these results reinforce the clue that commercial tools perform more powerful optimization during the technology mapping step.

Table 5.4: Synthetic ILB synthesis in a commercial tool

| I/O | AIG nodes | generic synthesis | mapped synthesis |
|---|---|---|---|
| 2 | 6 | 11 | 0 |
| 3 | 25 | 44 | 0 |
| 4 | 83 | 153 | 0 |
| 5 | 206 | 380 | 0 |
| 6 | 489 | 887 | 0 |
| 7 | 1,115 | 2,002 | 0 |
| 8 | 2,496 | 4,314 | 0 |
| 9 | 5,616 | 9,622 | 0 |
| 10 | 12,170 | 20,947 | 0 |
| 11 | 25,714 | 43,764 | 0 |
| 12 | 55,176 | 93,063 | 24,956 |
| 13 | 116,879 | 196,325 | 46,663 |
| 14 | 244,561 | 409,263 | 153,454 |
| 15 | 515,165 | 206,864 | 227,005 |

## 5.3 Embedding Custom Logic into ILB

In addition to generating the ILB by construction or deriving it from a real design, it is also possible to embed custom blocks, which may be useful to evaluate how algorithms act with respect to some function characteristic and structure.

Such a custom block may comprise a function whose circuit was exactly synthesized. It can be done with exact synthesis methods that run over small functions (SOEKEN et al., 2018)(SOEKEN et al., 2017)(HAASWIJK et al., 2018). Also, exact benchmarks proposed in (CONG; MINKOVICH, 2007) and in (AMARÚ et al., 2017) could be adopted as custom blocks.

By adding a custom block with an exact synthesized function into the ILB, it is possible to assess the noise caused by the redundant logic over the logic synthesis algorithm. That is, the main idea of adding a custom block with an exact synthesized function is to check if the logic synthesis tool is able to eliminate the redundant logic and recover the minimal solution.

There are several ways to place this custom block. For instance, the block can be placed as follows:

- Between primary inputs (PI) and the first stage of an ILB;
- Between the ILB second stage and the primary outputs (PO);
- Between any of the ILBs in a cascade.

It would also be possible to place a custom block between the ILB stages. However, notice that while putting the custom block between the two ILB stages, the block functionality may be lost. Let's address this issue considering a simple two I/O ILB, comprising two inverters, as illustrated by Figure 5.1.

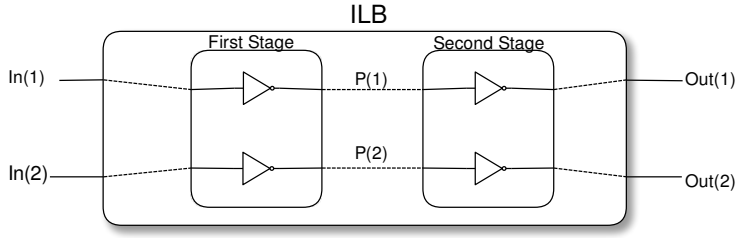
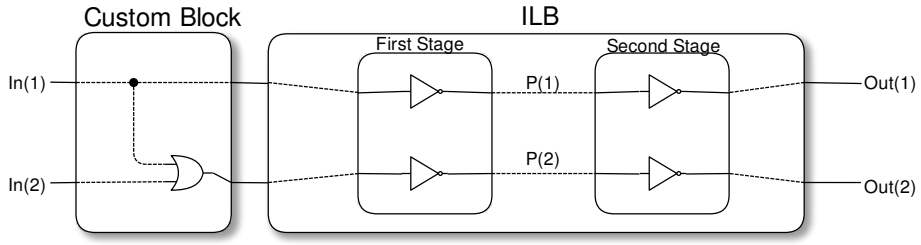
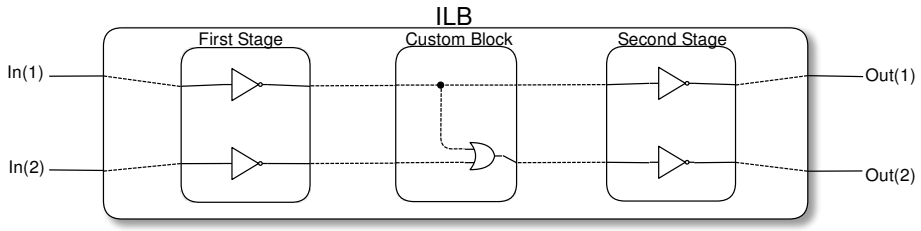



Figure 5.1: Example of an ILB composed by inverters.

By placing a custom block with the two input logic gate OR in front of the ILB, as presented in 5.2(a), the resulting circuit has $Out(1) = In(1)$ and $Out(2) = In(1) \vee In(2)$.




(a) Custom block placed between PIs and the ILB.




(b) Custom block between the ILB stages.

Figure 5.2: Example of custom block position in the ILB structure.

That is, the block functionality is keep and presented in $Out(2)$ whereas the $Out(1)$ keeps the value of $In(1)$.

However, when the custom block is moved to in between the ILB stages, as depicted in Figure 5.2(b) shows, the custom block (OR) behavior in the $Out(2)$ is lost. Consider, for instance, the case where $In(1) = 0, In(2) = 1$. Instead of having $Out(1) = 0, Out(2) = 1$, the $Out(2)$ is equals to 0. Therefore, placing custom blocks between ILB stages may lead to an unknown behavior.

To illustrate the potential of the custom block, a 2-input exclusive-OR (XOR2) gate was placed between the PIs and the ILB first stage, similar to illustrated in Figure 5.2(b) for the OR gate. We have opted by XOR2 gate because it is a more complex circuit than the OR one, and is represented by three AIG nodes. Therefore, its optimal implementation is known. In that sense, after running a logic synthesis algorithms over this ILB with this custom block, it is expected the final result to be three nodes.

Results with the XOR2 gate in our custom block are presented in Table 5.5. As can be seen in Table 5.5, the addition of a simple gate as custom logic may cause a small noise in the obtained results. For instance, the *drw* command has stopped finding the exact solution for the circuit with three I/O and 28 AIG nodes. Still, this experiment could be extended for exact multi-level functions.

Table 5.5: Results using XOR2 logic gate as custom logic.

| I/O | size | depth | *dc2* | *mig_rewrite* | *drw* | *compress2rs* | *dsd* |
|---|---|---|---|---|---|---|---|
| 2 | 9 | 6 | 0 | 9 | 0 | 0 | 0 |
| 3 | 28 | 10 | 0 | 27 | 11 | 0 | 0 |
| 4 | 86 | 14 | 0 | 81 | 0 | 0 | 0 |
| 5 | 209 | 19 | 146 | 206 | 170 | 146 | 0 |
| 6 | 492 | 24 | 354 | 486 | 395 | 346 | 0 |

## 5.4 Combining ILBs to increase circuit complexity

This section presents how ILBs can be used to increase the benchmark circuit complexity. To address that, we present two different possibilities: (i) arranging and combining ILBs, and (ii) using ILBs as redundant logic in other designs.

The first option allows to instantiate and combine as many ILBs as wished. To combine them, ILBs can be layered and cascaded. While layering ILBs, we are increasing the AIG breadth, being possible to simulate as many PI as wished. If one wants to embed a custom block with more PI than a single ILB, it is possible to instantiate as much ILBs as necessary to achieve the desired number of PI. On the other hand, by cascading ILBs and custom blocks the AIG depth is increased and logic depth oriented synthesis algorithms may be analyzed.

In this work, even though there is no limit in the number of ILBs connected, we present a two-layer connection, inspired in the LEKO circuits, as presented in Fig. 5.3. We have designed blocks layering ILB3, ILB4, ILB5, ILB6 and ILB7. They are refereed as H9, H16, H25, H36 and H49, respectively. To check if these arrangements impose some new challenges and difficulties to the ABC tool, they were also synthesized and the technology independent results are presented in Table 5.6. However, experimental results show that the tool is able to provide the exact solution when the arrangement is composed by ILBs that have their exact solution found when synthesized by their self. We believe that it happens because the tool is able to identify each block itself while running optimizations, so that each block is reduced to wires, leading to the exact global solution.

Results for this set of circuits after ASIC and FPGA technology mapping are presented in Table 5.7. It is worth to notice that the FPGA synthesis was able to found the exact solution for ILB5 and ILB6 but not for the respective H25 and H36 where they are replicated in an interconnected two-layer topology.

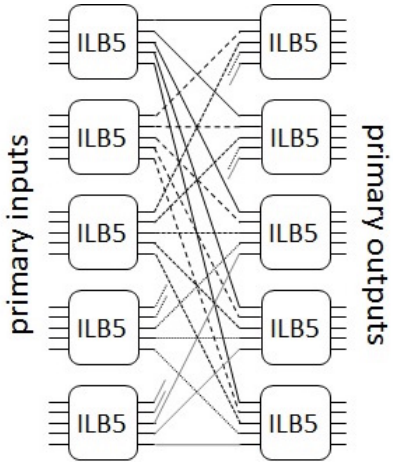



Figure 5.3: H25 ILB arrangement.

Table 5.6: Synthesis results for size (area) oriented ABC commands.

| Circuit | AIG nodes | dc2 | drw | compress2rs | dsd |
|---|---|---|---|---|---|
| H9 | 150 | 0 | 0 | 0 | 0 |
| H16 | 664 | 0 | 0 | 0 | 0 |
| H25 | 2,060 | 1,489 | 1,725 | 1,469 | 0 |
| H36 | 5,868 | 4,104 | 4,734 | 4,130 | 0 |
| H49 | 15,610 | 11,257 | 12,958 | 11,632 | 0 |

Finally, as for the second option, another set of experiments have been carried out aiming to verify the impact that identity blocks have inside other circuits. For that, C5 and G25 arrangements proposed by Cong and Minkovich, in (CONG; MINKOVICH, 2007), were described in Verilog and synthesized to act as reference in this further analysis. Notice that the G25 arrangement is a composition of two layers of C5, as depicted in Figure 3.1. The interleaved multi-level arrangements shown in Figure 5.4 have been taken into account. It was expected to obtain results for these mixed C5-ILB5 topologies equivalent to the G25 one, since such a transparent block (ILB circuit) does not include additional logic to the final circuit behavior, acting just as a sort of noise in the logic synthesis process. As observed in Table 5.8, it has not happened when comparing the synthesis results between M25a, M25b and M25c to the original G25 arrangement. As for the C5 circuit, one can notice that current FPGA tools were not able to find the optimal solution presented in (CONG; MINKOVICH, 2007), which comprises 70 LUTs. However, there are improvements when comparing the obtained results to the ones presented by Cong and Minkovich, in 2007.

Table 5.7: Benchmark synthesis in FPGA and ASIC design environments.

| Circuit | FPGA LUT | Std Cell |
|---|---|---|
| H9 | 0 | 0 |
| H16 | 0 | 0 |
| H25 | 26 | 444 |
| H36 | 76 | 1,752 |
| H49 | 3,170 | 3,469 |

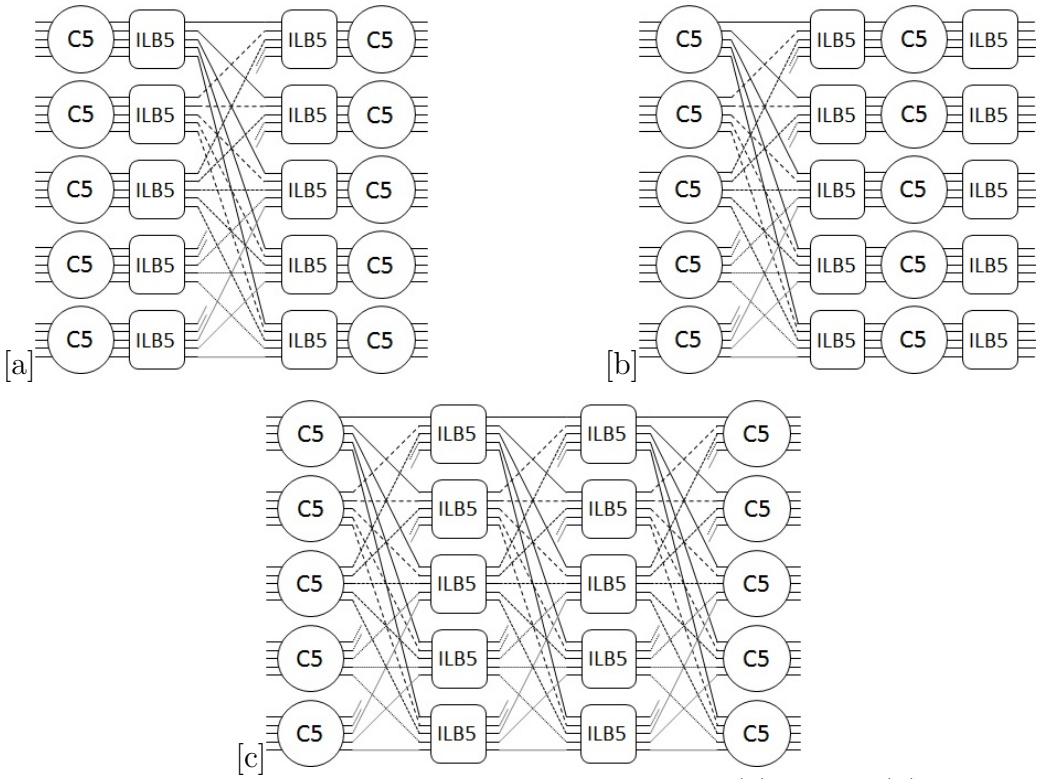



Figure 5.4: C5-ILB5 interleaved multi-layer arrangements: (a) M25a, (b) M25b and (c) M25c.

Table 5.8: Experimental results combining LEKO arrangements with ILBs.

| Circuit | FPGA LUT | Std Cell |
|---|---|---|
| C5 (CONG; MINKOVICH, 2007) | 7 | 13 |
| G25 (CONG; MINKOVICH, 2007) | 71 | 110 |
| M25a | 475 | 509 |
| M25b | 475 | 504 |
| M25c | 553 | 159 |

# 6 CONCLUSIONS

Logic synthesis methods rely on circuit benchmarks to evaluate its efficiency compared to previous approaches. The majority of publicly available benchmarks do not have an exact known solution, so when running methods over them it is not possible to know how well the current state-of-art logic synthesis algorithms are performing. On the other hand, by using exact multi-level logic circuits, it is possible to verify whether current methods are producing near-optimal solutions or if there is still room for further improvement.

Therefore, this work proposed a novel method to generate exact multi-level logic benchmarks circuits. The process is based on reversible logic and creates the circuits in two different ways. Firstly, open-source tools targeting reversible logic synthesis are explored, enabling the generation of exact benchmarks based on real designs. It means that the resulting circuit has portions of combinational logic computing real functions, being useful to tune algorithms concerning structures found in practical designs.

Secondly, this work presented an algorithm to generate exact synthetic circuits. Synthetic circuits are used to keep control on circuit size, being valuable to identify for which size of circuit tools start having difficulties. Also, it is possible to generate huge circuits through the synthetic approach, which can be used as a benchmark for future generations of logic synthesis tools. Both approaches presented to derive exact circuits in this work are challenging to current open-source and commercial tools.

The possibility of embedding custom blocks with specific logic were also discussed, which seems to be an interesting opportunity to asses methods in respect to functions with a given specificity. Moreover, the flexibility while arranging and connecting the generated blocks were discussed, so that blocks with more I/O can be created.

As future work, combinational equivalence checking (CEC) is seen as a promising application of the benchmarks proposed herein. CEC plays a significant role in EDA by verifying the functional equivalence between different combinational circuits after multi-level logic synthesis. That is, CEC tools must ascertain if two structurally different circuits have the same functionality.

Given the importance of CEC methods, they must be fast and should scale for large circuits. However, in the same way that multi-level logic synthesis, combinational equivalence checking is a complex task. In fact, it is a co-NP hard-problem (MISHCHENKO et al., 2006). In this sense, the increasing complexity of current designs may break down

current CEC tools (MISHCHENKO et al., 2006). Therefore, proper heuristic methods are are crucial (MATSUNAGA, 1996).

In that sense, ILBs may be a good candidate to increase complexity in designs and challenge CEC tools incrementally. By doing that, it would also be possible to know for which size of circuits these tools start to undesirable or even prohibitive execution time. Since ILBs implement identity function and do not change the logic, they could be inserted in any circuit, adding redundancy. For instance, a bus of eight wires could be connected to an eight I/O ILB. Therefore, additional redundant logic is attached and may disturb the verification tool.

Another possibility is to understand if it is possible to replace clouds of combinational logic by ILBs. That is not straightforward because extra inputs and outputs may be created while embedding the irreversible logic into a reversible function. Even though the additional inputs may be tied to a constant logic value (0, for instance), the extra outputs cannot be propagated to the POs to pass the combinational equivalence checking.

## REFERENCES

AKERS, S. B. Binary decision diagrams. **IEEE Transactions on computers**, IEEE, n. 6, p. 509–516, 1978.

ALBRECHT, C. Iwls 2005 benchmarks. In: INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS (IWLS). **Proceedings...** [S.l.: s.n.], 2005.

ALON, N.; BOPPANA, R. B. The monotone circuit complexity of boolean functions. **Combinatorica**, Springer, v. 7, n. 1, p. 1–22, 1987.

AMARÚ, L.; GAILLARDON, P.-E.; MICHELI, G. D. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In: ACM. PROCEEDINGS OF THE 51ST ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2014. p. 1–6.

AMARÚ, L.; GAILLARDON, P.-E.; MICHELI, G. D. The epfl combinational benchmark suite. In: INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS (IWLS). **Proceedings...** [S.l.: s.n.], 2015.

AMARÚ, L. et al. Multi-level logic benchmarks: An exactness study. In: IEEE. DESIGN AUTOMATION CONFERENCE (ASP-DAC), 2017 22ND ASIA AND SOUTH PACIFIC. **Proceedings...** [S.l.], 2017. p. 157–162.

AMARÚ, L. et al. Enabling exact delay synthesis. In: PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2017. (ICCAD '17), p. 352–359. Available from Internet: <http://dl.acm.org/citation.cfm?id=3199700.3199747>.

ASHENHURST, R. L. The decomposition of switching functions. In: PROCEEDINGS OF AN INTERNATIONAL SYMPOSIUM ON THE THEORY OF SWITCHING, APRIL 1957. **Proceedings...** [S.l.: s.n.], 1957.

BASTO, L. First results of itc'99 benchmark circuits. **IEEE Design Test of Computers**, v. 17, n. 3, p. 54–59, 2000.

BENNETT, C. H. Logical reversibility of computation. **IBM journal of Research and Development**, IBM, v. 17, n. 6, p. 525–532, 1973.

Berkeley Logic Synthesis and Verification Group. **ABC: A System for Sequential Synthesis and Verification**. 2018. Available from Internet: <http://www.eecs.berkeley.edu/~alanmi/abc/>. Accessed in: 2018-04-16.

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

BERTACCO, V.; OLUKOTUN, K. Efficient state representation for symbolic simulation. In: IEEE. DESIGN AUTOMATION CONFERENCE, 2002. PROCEEDINGS. 39TH. **Proceedings...** [S.l.], 2002. p. 99–104.

BÉRUT, A. et al. Experimental verification of landauer/'s principle linking information and thermodynamics. **Nature**, Nature Research, v. 483, n. 7388, p. 187–189, 2012.

BRAYTON, R.; MISHCHENKO, A. Abc: An academic industrial-strength verification tool. In: SPRINGER. INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION. **Proceedings...** [S.l.], 2010. p. 24–40.

BRAYTON, R. K. et al. **Logic minimization algorithms for VLSI synthesis**. [S.l.]: Springer Science & Business Media, 1984.

BRGLEZ, F.; BRYAN, D.; KOZMINSKI, K. Combinational profiles of sequential benchmark circuits. In: CIRCUITS AND SYSTEMS, 1989., IEEE INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.: s.n.], 1989.

BRGLEZ, F.; FUJIWARA, H. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran. In: PROC. OF THE INT'L SYMPOSIUM CIRCUITS AND SYSTEMS (ISCAS). **Proceedings...** [S.l.: s.n.], 1985. p. 677–692.

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **Computers, IEEE Transactions on**, IEEE, v. 100, n. 8, p. 677–691, 1986.

CHANG, C.-C. et al. Optimality and scalability study of existing placement algorithms. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 23, n. 4, p. 537–549, 2004.

CHATTERJEE, S. et al. Reducing structural bias in technology mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 25, n. 12, p. 2894–2903, 2006.

CHEN, D. et al. Fpga design automation: A survey. **Foundations and Trends® in Electronic Design Automation**, Now Publishers, Inc., v. 1, n. 3, p. 195–330, 2006.

CONG, J.; MINKOVICH, K. Optimality study of logic synthesis for lut-based fpgas. **IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems**, IEEE, v. 26, n. 2, p. 230–239, 2007.

CORNO, F.; REORDA, M.; SQUILLERO, G. Rt-level itc'99 benchmarks and first atpg results. **Design Test of Computers, IEEE**, v. 17, n. 3, p. 44–53, Jul 2000.

COUDERT, O. Two-level logic minimization: an overview. **Integration, the VLSI journal**, Elsevier, v. 17, n. 2, p. 97–140, 1994.

CURTIS, H. A. **A new approach to the design of switching circuits**. [S.l.]: van Nostrand, 1962.

DAVIDSON, S.; HARLOW, J. Guest editors' introduction: Benchmarking for design and test. **IEEE Design & Test**, IEEE Computer Society Press, v. 17, n. 3, p. 12–14, 2000.

ERNST, E. A. **Optimal combinational multi-level logic synthesis**. Thesis (PhD) — University of Michigan, 2009.

FLEISHER, H.; MAISSEL, L. I. An introduction to array logic. **IBM Journal of Research and Development**, IBM, v. 19, n. 2, p. 98–109, 1975.

HAASWIJK, W. et al. Sat based exact synthesis using dag topology families. In: DAC. **Proceedings...** [S.l.: s.n.], 2018. p. 53–1.

HACHTEL, G. D.; SOMENZI, F. **Logic synthesis and verification algorithms**. [S.l.]: Springer Science & Business Media, 2006.

HARLOW, J. E. Overview of popular benchmark sets. **IEEE Design & Test of Computers**, IEEE, v. 17, n. 3, p. 15–17, 2000.

KLEIN, J. P.; LEETE, T. H.; RUBIN, H. A biomolecular implementation of logically reversible computation with minimal energy dissipation. **Biosystems**, Elsevier, v. 52, n. 1, p. 15–23, 1999.

KNILL, E.; LAFLAMME, R.; MILBURN, G. J. A scheme for efficient quantum computation with linear optics. **nature**, Nature Publishing Group, v. 409, n. 6816, p. 46–52, 2001.

KUEHLMANN, A.; KROHM, F. Equivalence checking using cuts and heaps. In: ACM. PROCEEDINGS OF THE 34TH ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 1997. p. 263–268.

KUTZSCHEBAUCH, T.; STOK, L. Layout driven decomposition with congestion consideration. In: IEEE COMPUTER SOCIETY. PROCEEDINGS OF THE CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE. **Proceedings...** [S.l.], 2002. p. 672.

LANDAUER, R. Irreversibility and heat generation in the computing process. **IBM journal of research and development**, Ibm, v. 5, n. 3, p. 183–191, 1961.

Mathias Soeken. **CirKit**. 2018. Release 20130425. Available from Internet: <https://msoeken.github.io/cirkit.html>. Accessed in: 2018-04-16.

MATSUNAGA, Y. An efficient equivalence checker for combinational circuits. In: ACM. PROCEEDINGS OF THE 33RD ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 1996. p. 629–634.

MCELVAIN, K. Iwls'93 benchmark set: Version 4.0. In: INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS (IWLS). **Proceedings...** [S.l.: s.n.], 1993.

MCGEER, P. C. et al. Espresso-signature: A new exact minimizer for logic functions. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 1, n. 4, p. 432–440, 1993.

MICHELI, G. D. **Synthesis and optimization of digital circuits**. [S.l.]: McGraw-Hill Higher Education, 1994.

MISHCHENKO, A.; BRAYTON, R. Faster logic manipulation for large designs. Citeseer, 2007.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In: ACM. PROCEEDINGS OF THE 43RD ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2006. p. 532–535.

MISHCHENKO, A. et al. Improvements to combinational equivalence checking. In: ACM. PROCEEDINGS OF THE 2006 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 2006. p. 836–843.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. K. Improvements to technology mapping for lut-based fpgas. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 26, n. 2, p. 240–253, 2007.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. K. Improvements to technology mapping for lut-based fpgas. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 26, n. 2, p. 240–253, 2007.

MOORE, G. Moore's law. **Electronics Magazine**, v. 38, n. 8, p. 114, 1965.

NIELSEN, M. A.; CHUANG, I. **Quantum computation and quantum information**. [S.l.]: AAPT, 2002.

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

PISTORIUS, J. et al. Benchmarking method and designs targeting logic synthesis for fpgas. In: INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS (IWLS). **Proceedings...** [S.l.: s.n.], 2007.

PLAZA, S.; BERTACCO, V. Staccato: disjoint support decompositions from bdds through symbolic kernels. In: ACM. PROCEEDINGS OF THE 2005 ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2005. p. 276–279.

RABAEY, J. M.; CHANDRAKASAN, A. P.; NIKOLIC, B. **Digital integrated circuits**. [S.l.]: Prentice hall Englewood Cliffs, 2002.

SAEEDI, M.; MARKOV, I. L. Synthesis and optimization of reversible circuits—a survey. **ACM Computing Surveys (CSUR)**, ACM, v. 45, n. 2, p. 21, 2013.

Scott Davidson. **ITC'99 Benchmarks**. 1999. Available from Internet: <https://www.cerc.utexas.edu/itc99-benchmarks/bendoc1.html>. Accessed in: 2018-04-25.

SENTOVICH, E. M. et al. SIS: A system for sequential circuit synthesis. Univ. California, Berkeley, Tech. Rep. UCB/ERL M92/41, 1992.

SOEKEN, M. et al. Exact synthesis of majority-inverter graphs and its applications. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 36, n. 11, p. 1842–1855, 2017.

SOEKEN, M. et al. RevKit: An open source toolkit for the design of reversible circuits. In: REVERSIBLE COMPUTATION 2011. **Proceedings...** [S.l.: s.n.], 2012. (Lecture Notes in Computer Science, v. 7165), p. 64–76. RevKit is available at www.revkit.org.

SOEKEN, M. et al. Practical exact synthesis. In: IEEE. DESIGN, AUTOMATION & TEST IN EUROPE CONFERENCE & EXHIBITION (DATE), 2018. **Proceedings...** [S.l.], 2018. p. 309–314.

SOEKEN, M. et al. Sat-based combinational and sequential dependency computation. In: SPRINGER. HAIFA VERIFICATION CONFERENCE. **Proceedings...** [S.l.], 2016. p. 1–17.

SOEKEN, M. et al. Embedding of large boolean functions for reversible logic. **ACM Journal on Emerging Technologies in Computing Systems (JETC)**, ACM, v. 12, n. 4, p. 41, 2016.

STOK, L. Developing parallel eda tools [the last byte]. **IEEE Design & Test**, IEEE, v. 30, n. 1, p. 65–66, 2013.

THORP, T.; YEE, G.; SECHEN, C. Design and synthesis of monotonic circuits. In: IEEE. COMPUTER DESIGN, 1999.(ICCD'99) INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.], 1999. p. 569–572.

WESTE, N.; HARRIS, D.; BANERJEE, A. Cmos vlsi design. **A circuits and systems perspective**, v. 11, p. 739, 2005.

WILLE, R.; DRECHSLER, R. **Towards a design flow for reversible logic**. [S.l.]: Springer Science & Business Media, 2010.

YANG, S. Logic synthesis and optimization benchmarks. In: INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS (IWLS). **Proceedings...** [S.l.: s.n.], 1989.

YANG, S. **Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0**. [S.l.], 1991.

YANG, S. Logic synthesis and optimization benchmarks version 3.0. **Tech. Report, Microelectronics Centre of North Carolina**, 1991.

ZULEHNER, A.; WILLE, R. Make it reversible: Efficient embedding of non-reversible functions. In: IEEE. 2017 DESIGN, AUTOMATION & TEST IN EUROPE CONFERENCE & EXHIBITION (DATE). **Proceedings...** [S.l.], 2017. p. 458–463.